

Copyright

Copyright ? Made by the Asta Xie. This material may be distributed only subject to the terms and conditions set forth in the Creative Commons Attribution 3.0 License or later. A copy of the [Creative Commons Attribution 3.0 license](http://creativecommons.org/licenses/by/3.0/) is distributed with this manual. The latest version is presently available at [? http://creativecommons.org/licenses/by/3.0/](http://creativecommons.org/licenses/by/3.0/).

If you are interested in redistribution or republishing of this document in whole or in part, either modified or unmodified, and you have questions, please contact the Copyright holders at xiemengjun@gmail.com.

The CHM Manual is Made By AstaXie

xiemengjun@gmail.com

Go is an expressive, concurrent, garbage-collected programming language.

The [Go home page](#) is the primary source of information about Go. It contains [installation instructions](#), [a tutorial](#), and more.

This repository holds the gc compilers and Go packages. Changes to the code are [reviewed](#) before being committed.

The gccgo front end for GCC is hosted as a branch on the GCC Subversion server. See [the gccgo installation instructions](#).



Getting Started

Introduction

Go is an open source project with a BSD-style license. There are two official Go compiler toolchains: the gc Go compiler and the gccgo compiler that is part of the GNU C Compiler (GCC).

The gc compiler is the more mature and well-tested of the two. This page is about installing a binary distribution of the gc compiler.

For information about installing the gc compiler from source, see [Installing Go from source](#). For information about installing gccgo, see [Setting up and using gccgo](#).

Download the Go tools

Visit the [Go project's downloads page](#) and select the binary distribution that matches your operating system and processor architecture.

Official binary distributions are available for the FreeBSD, Linux, Mac OS X (Snow Leopard/Lion), and Windows operating systems and the 32-bit (386) and 64-bit (amd64) x86 processor architectures.

If a binary distribution is not available for your OS/arch combination you may want to try [installing from source](#) or [installing gccgo instead of gc](#).

Install the Go tools

The Go binary distributions assume they will be installed in `/usr/local/go` (or `c:\Go` under Windows), but it is possible to install them in a different location. If you do this, you will need to set the `GOROOT` environment variable to that directory when using the Go tools.

For example, if you installed Go to your home directory you should add the following commands to `$HOME/.profile`:

```
export GOROOT=$HOME/go
export PATH=$PATH:$GOROOT/bin
```

Windows users should read the section about [setting environment variables under Windows](#).

FreeBSD and Linux

On FreeBSD and Linux, if you are upgrading from an older version of Go you must first remove the existing version from `/usr/local/go`:

```
rm -r /usr/local/go
```

Extract [the archive](#) into `/usr/local`, creating a Go tree in `/usr/local/go`:

```
tar -C /usr/local -xzf go.release.go1.tar.gz
```

(Typically these commands must be run as root or through `sudo`.)

Add `/usr/local/go/bin` to the `PATH` environment variable. You can do this by adding this line to your `/etc/profile` (for a system-wide installation) or `$HOME/.profile`:

```
export PATH=$PATH:/usr/local/go/bin
```

Mac OS X

Open the [package file](#) and follow the prompts to install the Go tools. The package installs the Go distribution to `/usr/local/go`.

The package should put the `/usr/local/go/bin` directory in your `PATH` environment variable. You may need to restart any open Terminal sessions for the change to take effect.

Windows

The Go project provides two installation options for Windows users (besides [installing from source](#)): a zip archive that requires you to set some environment variables and an experimental MSI installer that configures your installation automatically.

Zip archive

Extract the [zip file](#) to the directory of your choice (we suggest `c:\Go`).

If you chose a directory other than `c:\Go`, you must set the `GOROOT` environment variable to your chosen path.

Add the `bin` subdirectory of your Go root (for example, `c:\Go\bin`) to to your `PATH` environment variable.

MSI installer (experimental)

Open the [MSI file](#) and follow the prompts to install the Go tools. By default, the installer puts the Go distribution in `c:\Go`.

The installer should put the `c:\Go\bin` directory in your `PATH` environment variable. You may need to restart any open command prompts for the change to take effect.

Setting environment variables under Windows

Under Windows, you may set environment variables through the "Environment Variables" button on the "Advanced" tab of the "System" control panel. Some versions of Windows provide this control panel through the "Advanced System Settings" option inside the "System" control panel.

Test your installation

Check that Go is installed correctly by building a simple program, as follows.

Create a file named `hello.go` and put the following program in it:

```
package main

import "fmt"

func main() {
    fmt.Printf("hello, world\n")
}
```

Then run it with the go tool:

```
$ go run hello.go
hello, world
```

If you see the "hello, world" message then your Go installation is working.

What's next

Start by taking [A Tour of Go](#).

For more detail about the process of building and testing Go programs read [How to Write Go Code](#).

Build a web application by following the [Wiki Tutorial](#).

Read [Effective Go](#) to learn about writing idiomatic Go code.

For the full story, consult Go's extensive [documentation](#).

Subscribe to the [golang-announce](#) mailing list to be notified when a new stable version of Go is released.

Community resources

For real-time help, there may be users or developers on #go-nuts on the [Freenode](#) IRC server.

The official mailing list for discussion of the Go language is [Go Nuts](#).

Bugs should be reported using the [Go issue tracker](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

How to Write Go Code

Introduction

This document demonstrates the development of a simple Go package and introduces the [go command](#), the standard way to fetch, build, and install Go packages and commands.

Code organization

GOPATH and workspaces

One of Go's design goals is to make writing software easier. To that end, the `go` command doesn't use Makefiles or other configuration files to guide program construction. Instead, it uses the source code to find dependencies and determine build conditions. This means your source code and build scripts are always in sync; they are one and the same.

The one thing you must do is set a `GOPATH` environment variable. `GOPATH` tells the `go` command (and other related tools) where to find and install the Go packages on your system.

`GOPATH` is a list of paths. It shares the syntax of your system's `PATH` environment variable. A typical `GOPATH` on a Unix system might look like this:

```
GOPATH=/home/user/ext:/home/user/mygo
```

(On a Windows system use semicolons as the path separator instead of colons.)

Each path in the list (in this case `/home/user/ext` or `/home/user/mygo`) specifies the location of a *workspace*. A workspace contains Go source files and their associated package objects, and command executables. It has a prescribed structure of three subdirectories:

- `src` contains Go source files,
- `pkg` contains compiled package objects, and
- `bin` contains executable commands.

Subdirectories of the `src` directory hold independent packages, and all source files (`.go`, `.c`, `.h`, and `.s`) in each subdirectory are elements of that subdirectory's package.

When building a program that imports the package "widget" the `go` command looks for `src/pkg/widget` inside the Go root, and then—if the package source isn't found there—it searches for `src/widget` inside each workspace in order.

Multiple workspaces can offer some flexibility and convenience, but for now we'll concern ourselves with only a single workspace.

Let's work through a simple example. First, create a `$HOME/mygo` directory and its `src` subdirectory:

```
$ mkdir -p $HOME/mygo/src # create a place to put source code
```

Next, set it as the `GOPATH`. You should also add the `bin` subdirectory to your `PATH` environment variable so that you can run the commands therein without specifying their full path. To do this, add the following lines to `$HOME/.profile` (or equivalent):

```
export GOPATH=$HOME/mygo
export PATH=$PATH:$HOME/mygo/bin
```

Import paths

The standard packages are given short import paths such as `"fmt"` and `"net/http"` for convenience. For your own projects, it is important to choose a base import path that is unlikely to collide with future additions to the standard library or other external libraries.

The best way to choose an import path is to use the location of your version control repository. For instance, if your source repository is at `example.com` or `code.google.com/p/example`, you should begin your package paths with that URL, as in `"example.com/foo/bar"` or `"code.google.com/p/example/foo/bar"`. Using this convention, the `go` command can automatically check out and build the source code by its import path alone.

If you don't intend to install your code in this way, you should at least use a unique prefix like `"widgets/"`, as in `"widgets/foo/bar"`. A good rule is to use a prefix such as your company or project name, since it is unlikely to be used by another group.

We'll use `example/` as our base import path:

```
$ mkdir -p $GOPATH/src/example
```

Package names

The first statement in a Go source file should be

```
package name
```

where *name* is the package's default name for imports. (All files in a package must use the same *name*.)

Go's convention is that the package name is the last element of the import path: the package imported as "crypto/rot13" should be named rot13. There is no requirement that package names be unique across all packages linked into a single binary, only that the import paths (their full file names) be unique.

Create a new package under example called newmath:

```
$ cd $GOPATH/src/example  
$ mkdir newmath
```

Then create a file named \$GOPATH/src/example/newmath/sqrt.go containing the following Go code:

```
// Package newmath is a trivial example package.  
package newmath  
  
// Sqrt returns an approximation to the square root of x.  
func Sqrt(x float64) float64 {  
    // This is a terrible implementation.  
    // Real code should import "math" and use math.Sqrt.  
    z := 0.0  
    for i := 0; i < 1000; i++ {  
        z -= (z*z - x) / (2 * x)  
    }  
    return z  
}
```

This package is imported by the path name of the directory it's in, starting after the src component:

```
import "example/newmath"
```

See [Effective Go](#) to learn more about Go's naming conventions.

Building and installing

The `go` command comprises several subcommands, the most central being `install`. Running `go install importpath` builds and installs a package and its dependencies.

To "install a package" means to write the package object or executable command to the `pkg` or `bin` subdirectory of the workspace in which the source resides.

Building a package

To build and install the `newmath` package, type

```
$ go install example/newmath
```

This command will produce no output if the package and its dependencies are built and installed correctly.

As a convenience, the `go` command will assume the current directory if no `import path` is specified on the command line. This sequence of commands has the same effect as the one above:

```
$ cd $GOPATH/src/example/newmath
$ go install
```

The resulting workspace directory tree (assuming we're running Linux on a 64-bit system) looks like this:

```
pkg/
  linux_amd64/
    example/
      newmath.a # package object
src/
  example/
    newmath/
      sqrt.go # package source
```

Building a command

The `go` command treats code belonging to package `main` as an executable

command and installs the package binary to the GOPATH's bin subdirectory.

Add a command named `hello` to the source tree. First create the `example/hello` directory:

```
$ cd $GOPATH/src/example
$ mkdir hello
```

Then create the file `$GOPATH/src/example/hello/hello.go` containing the following Go code.

```
// Hello is a trivial example of a main package.
package main

import (
    "example/newmath"
    "fmt"
)

func main() {
    fmt.Printf("Hello, world.  Sqrt(2) = %v\n", newmath.Sqrt(2))
}
```

Next, run `go install`, which builds and installs the binary to `$GOPATH/bin` (or `$GOBIN`, if set; to simplify presentation, this document assumes `GOBIN` is unset):

```
$ go install example/hello
```

To run the program, invoke it by name as you would any other command:

```
$ $GOPATH/bin/hello
Hello, world.  Sqrt(2) = 1.414213562373095
```

If you added `$HOME/mygo/bin` to your `PATH`, you may omit the path to the executable:

```
$ hello
Hello, world.  Sqrt(2) = 1.414213562373095
```

The workspace directory tree now looks like this:

```
bin/
  hello          # command executable
pkg/
  linux_amd64/
```

```
    example/  
        newmath.a # package object  
src/  
    example/  
        hello/  
            hello.go # command source  
        newmath/  
            sqrt.go # package source
```

The `go` command also provides a `build` command, which is like `install` except it builds all objects in a temporary directory and does not install them under `pkg` or `bin`. When building a command an executable named after the last element of the import path is written to the current directory. When building a package, `go build` serves merely to test that the package and its dependencies can be built. (The resulting package object is thrown away.)

Testing

Go has a lightweight test framework composed of the `go test` command and the `testing` package.

You write a test by creating a file with a name ending in `_test.go` that contains functions named `TestXXX` with signature `func (t *testing.T)`. The test framework runs each such function; if the function calls a failure function such as `t.Error` or `t.Fail`, the test is considered to have failed.

Add a test to the `newmath` package by creating the file `$GOPATH/src/example/newmath/sqrt_test.go` containing the following Go code.

```
package newmath

import "testing"

func TestSqrt(t *testing.T) {
    const in, out = 4, 2
    if x := Sqrt(in); x != out {
        t.Errorf("Sqrt(%v) = %v, want %v", in, x, out)
    }
}
```

Now run the test with `go test`:

```
$ go test example/newmath
ok      example/newmath 0.165s
```

Run [go help test](#) and see the [testing package documentation](#) for more detail.

Remote packages

An import path can describe how to obtain the package source code using a revision control system such as Git or Mercurial. The `go` command uses this property to automatically fetch packages from remote repositories. For instance, the examples described in this document are also kept in a Mercurial repository hosted at Google Code, code.google.com/p/go.example. If you include the repository URL in the package's import path, `go get` will fetch, build, and install it automatically:

```
$ go get code.google.com/p/go.example/hello
$ $GOPATH/bin/hello
Hello, world.  Sqrt(2) = 1.414213562373095
```

If the specified package is not present in a workspace, `go get` will place it inside the first workspace specified by `GOPATH`. (If the package does already exist, `go get` skips the remote fetch and behaves the same as `go install`.)

After issuing the above `go get` command, the workspace directory tree should now look like this:

```
bin/
  hello          # command executable
pkg/
  linux_amd64/
    code.google.com/p/go.example/
      newmath.a  # package object
    example/
      newmath.a  # package object
src/
  code.google.com/p/go.example/
    hello/
      hello.go   # command source
    newmath/
      sqrt.go    # package source
      sqrt_test.go # test source
  example/
    hello/
      hello.go   # command source
    newmath/
      sqrt.go    # package source
      sqrt_test.go # test source
```

The `hello` command hosted at Google Code depends on the `newmath` package within the same repository. The imports in `hello.go` file use the same import path convention, so the `go get` command is able to locate and install the dependent package, too.

```
import "code.google.com/p/go.example/newmath"
```

This convention is the easiest way to make your Go packages available for others to use. The [Go Project Dashboard](#) is a list of external Go projects including programs and libraries.

For more information on using remote repositories with the `go` command, see [go help remote](#).

Further reading

See [Effective Go](#) for tips on writing clear, idiomatic Go code.

Take [A Tour of Go](#) to learn the language proper.

Visit the [documentation page](#) for a set of in-depth articles about the Go language and its libraries and tools.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Effective Go

Introduction

Go is a new language. Although it borrows ideas from existing languages, it has unusual properties that make effective Go programs different in character from programs written in its relatives. A straightforward translation of a C++ or Java program into Go is unlikely to produce a satisfactory result—Java programs are written in Java, not Go. On the other hand, thinking about the problem from a Go perspective could produce a successful but quite different program. In other words, to write Go well, it's important to understand its properties and idioms. It's also important to know the established conventions for programming in Go, such as naming, formatting, program construction, and so on, so that programs you write will be easy for other Go programmers to understand.

This document gives tips for writing clear, idiomatic Go code. It augments the [language specification](#), the [Tour of Go](#), and [How to Write Go Code](#), all of which you should read first.

Examples

The [Go package sources](#) are intended to serve not only as the core library but also as examples of how to use the language. If you have a question about how to approach a problem or how something might be implemented, they can provide answers, ideas and background.

Formatting

Formatting issues are the most contentious but the least consequential. People can adapt to different formatting styles but it's better if they don't have to, and less time is devoted to the topic if everyone adheres to the same style. The problem is how to approach this Utopia without a long prescriptive style guide.

With Go we take an unusual approach and let the machine take care of most formatting issues. The `gofmt` program (also available as `go fmt`, which operates at the package level rather than source file level) reads a Go program and emits the source in a standard style of indentation and vertical alignment, retaining and if necessary reformatting comments. If you want to know how to handle some new layout situation, run `gofmt`; if the answer doesn't seem right, rearrange your program (or file a bug about `gofmt`), don't work around it.

As an example, there's no need to spend time lining up the comments on the fields of a structure. `gofmt` will do that for you. Given the declaration

```
type T struct {
    name string // name of the object
    value int // its value
}
```

`gofmt` will line up the columns:

```
type T struct {
    name    string // name of the object
    value   int    // its value
}
```

All Go code in the standard packages has been formatted with `gofmt`.

Some formatting details remain. Very briefly,

Indentation

We use tabs for indentation and `gofmt` emits them by default. Use spaces only if you must.

Line length

Go has no line length limit. Don't worry about overflowing a punched card. If a line feels too long, wrap it and indent with an extra tab.

Parentheses

Go needs fewer parentheses: control structures (`if`, `for`, `switch`) do not have parentheses in their syntax. Also, the operator precedence hierarchy is shorter and clearer, so

```
x<<8 + y<<16
```

means what the spacing implies.

Commentary

Go provides C-style `/* */` block comments and C++-style `//` line comments. Line comments are the norm; block comments appear mostly as package comments and are also useful to disable large swaths of code.

The program `godoc` and web server `godoc` processes Go source files to extract documentation about the contents of the package. Comments that appear before top-level declarations, with no intervening newlines, are extracted along with the declaration to serve as explanatory text for the item. The nature and style of these comments determines the quality of the documentation `godoc` produces.

Every package should have a *package comment*, a block comment preceding the package clause. For multi-file packages, the package comment only needs to be present in one file, and any one will do. The package comment should introduce the package and provide information relevant to the package as a whole. It will appear first on the `godoc` page and should set up the detailed documentation that follows.

```
/*
    Package regexp implements a simple library for
    regular expressions.

    The syntax of the regular expressions accepted is:

    regexp:
        concatenation { '|' concatenation }
    concatenation:
        { closure }
    closure:
        term [ '*' | '+' | '?' ]
    term:
        '^'
        '$'
        '.'
        character
        '[' [ '^' ] character-ranges '['
        '(' regexp ')'
*/
package regexp
```

If the package is simple, the package comment can be brief.

```
// Package path implements utility routines for
// manipulating slash-separated filename paths.
```

Comments do not need extra formatting such as banners of stars. The generated output may not even be presented in a fixed-width font, so don't depend on spacing for alignment—`godoc`, like `gofmt`, takes care of that. The comments are uninterpreted plain text, so HTML and other annotations such as `_this_` will reproduce *verbatim* and should not be used. Depending on the context, `godoc` might not even reformat comments, so make sure they look good straight up: use correct spelling, punctuation, and sentence structure, fold long lines, and so on.

Inside a package, any comment immediately preceding a top-level declaration serves as a *doc comment* for that declaration. Every exported (capitalized) name in a program should have a doc comment.

Doc comments work best as complete sentences, which allow a wide variety of automated presentations. The first sentence should be a one-sentence summary that starts with the name being declared.

```
// Compile parses a regular expression and returns, if successful, a
// object that can be used to match against text.
func Compile(str string) (regexp *Regexp, err error) {
```

Go's declaration syntax allows grouping of declarations. A single doc comment can introduce a group of related constants or variables. Since the whole declaration is presented, such a comment can often be perfunctory.

```
// Error codes returned by failures to parse an expression.
var (
    ErrInternal          = errors.New("regexp: internal error")
    ErrUnmatchedLpar    = errors.New("regexp: unmatched '('")
    ErrUnmatchedRpar    = errors.New("regexp: unmatched ')'")
    ...
)
```

Even for private names, grouping can also indicate relationships between items, such as the fact that a set of variables is protected by a mutex.

```
var (
    countLock    sync.Mutex
    inputCount   uint32
    outputCount  uint32
    errorCount   uint32
)
```

Names

Names are as important in Go as in any other language. In some cases they even have semantic effect: for instance, the visibility of a name outside a package is determined by whether its first character is upper case. It's therefore worth spending a little time talking about naming conventions in Go programs.

Package names

When a package is imported, the package name becomes an accessor for the contents. After

```
import "bytes"
```

the importing package can talk about `bytes.Buffer`. It's helpful if everyone using the package can use the same name to refer to its contents, which implies that the package name should be good: short, concise, evocative. By convention, packages are given lower case, single-word names; there should be no need for underscores or mixedCaps. Err on the side of brevity, since everyone using your package will be typing that name. And don't worry about collisions *a priori*. The package name is only the default name for imports; it need not be unique across all source code, and in the rare case of a collision the importing package can choose a different name to use locally. In any case, confusion is rare because the file name in the import determines just which package is being used.

Another convention is that the package name is the base name of its source directory; the package in `src/pkg/encoding/base64` is imported as `"encoding/base64"` but has name `base64`, not `encoding_base64` and not `encodingBase64`.

The importer of a package will use the name to refer to its contents (the `import .` notation is intended mostly for tests and other unusual situations and should be avoided unless necessary), so exported names in the package can use that fact to avoid stutter. For instance, the buffered reader type in the `bufio` package is called `Reader`, not `BufReader`, because users see it as `bufio.Reader`, which is a clear, concise name. Moreover, because imported entities are always addressed with their package name, `bufio.Reader` does not conflict with `io.Reader`. Similarly, the function to make new instances of `ring.Ring`—which is the

definition of a *constructor* in Go—would normally be called `NewRing`, but since `Ring` is the only type exported by the package, and since the package is called `ring`, it's called just `New`, which clients of the package see as `ring.New`. Use the package structure to help you choose good names.

Another short example is `once.Do`; `once.Do(setup)` reads well and would not be improved by writing `once.DoOrwaitUntilDone(setup)`. Long names don't automatically make things more readable. If the name represents something intricate or subtle, it's usually better to write a helpful doc comment than to attempt to put all the information into the name.

Getters

Go doesn't provide automatic support for getters and setters. There's nothing wrong with providing getters and setters yourself, and it's often appropriate to do so, but it's neither idiomatic nor necessary to put `Get` into the getter's name. If you have a field called `owner` (lower case, unexported), the getter method should be called `Owner` (upper case, exported), not `GetOwner`. The use of upper-case names for export provides the hook to discriminate the field from the method. A setter function, if needed, will likely be called `SetOwner`. Both names read well in practice:

```
owner := obj.Owner()
if owner != user {
    obj.SetOwner(user)
}
```

Interface names

By convention, one-method interfaces are named by the method name plus the `-er` suffix: `Reader`, `Writer`, `Formatter` etc.

There are a number of such names and it's productive to honor them and the function names they capture. `Read`, `Write`, `Close`, `Flush`, `String` and so on have canonical signatures and meanings. To avoid confusion, don't give your method one of those names unless it has the same signature and meaning. Conversely, if your type implements a method with the same meaning as a method on a well-known type, give it the same name and signature; call your string-converter method `String` not `ToString`.

MixedCaps

Finally, the convention in Go is to use `MixedCaps` or `mixedCaps` rather than underscores to write multiword names.

Semicolons

Like C, Go's formal grammar uses semicolons to terminate statements; unlike C, those semicolons do not appear in the source. Instead the lexer uses a simple rule to insert semicolons automatically as it scans, so the input text is mostly free of them.

The rule is this. If the last token before a newline is an identifier (which includes words like `int` and `float64`), a basic literal such as a number or string constant, or one of the tokens

```
break continue fallthrough return ++ -- ) }
```

the lexer always inserts a semicolon after the token. This could be summarized as, “if the newline comes after a token that could end a statement, insert a semicolon”.

A semicolon can also be omitted immediately before a closing brace, so a statement such as

```
go func() { for { dst <- <-src } }()
```

needs no semicolons. Idiomatic Go programs have semicolons only in places such as `for` loop clauses, to separate the initializer, condition, and continuation elements. They are also necessary to separate multiple statements on a line, should you write code that way.

One caveat. You should never put the opening brace of a control structure (`if`, `for`, `switch`, or `select`) on the next line. If you do, a semicolon will be inserted before the brace, which could cause unwanted effects. Write them like this

```
if i < f() {  
    g()  
}
```

not like this

```
if i < f() // wrong!  
{ // wrong!  
    g()
```

}

Control structures

The control structures of Go are related to those of C but differ in important ways. There is no `do` or `while` loop, only a slightly generalized `for`; `switch` is more flexible; `if` and `switch` accept an optional initialization statement like that of `for`; and there are new control structures including a type switch and a multiway communications multiplexer, `select`. The syntax is also slightly different: there are no parentheses and the bodies must always be brace-delimited.

If

In Go a simple `if` looks like this:

```
if x > 0 {
    return y
}
```

Mandatory braces encourage writing simple `if` statements on multiple lines. It's good style to do so anyway, especially when the body contains a control statement such as a `return` or `break`.

Since `if` and `switch` accept an initialization statement, it's common to see one used to set up a local variable.

```
if err := file.Chmod(0664); err != nil {
    log.Print(err)
    return err
}
```

In the Go libraries, you'll find that when an `if` statement doesn't flow into the next statement `return` that is, the body ends in `break`, `continue`, `goto`, or `return` the unnecessary `else` is omitted.

```
f, err := os.Open(name)
if err != nil {
    return err
}
codeUsing(f)
```

This is an example of a common situation where code must guard against a

sequence of error conditions. The code reads well if the successful flow of control runs down the page, eliminating error cases as they arise. Since error cases tend to end in return statements, the resulting code needs no else statements.

```
f, err := os.Open(name)
if err != nil {
    return err
}
d, err := f.Stat()
if err != nil {
    f.Close()
    return err
}
codeUsing(f, d)
```

Redeclaration

An aside: The last example in the previous section demonstrates a detail of how the := short declaration form works. The declaration that calls os.Open reads,

```
f, err := os.Open(name)
```

This statement declares two variables, f and err. A few lines later, the call to f.Stat reads,

```
d, err := f.Stat()
```

which looks as if it declares d and err. Notice, though, that err appears in both statements. This duplication is legal: err is declared by the first statement, but only *re-assigned* in the second. This means that the call to f.Stat uses the existing err variable declared above, and just gives it a new value.

In a := declaration a variable v may appear even if it has already been declared, provided:

- this declaration is in the same scope as the existing declaration of v (if v is already declared in an outer scope, the declaration will create a new variable),
- the corresponding value in the initialization is assignable to v, and
- there is at least one other variable in the declaration that is being declared anew.

This unusual property is pure pragmatism, making it easy to use a single `err` value, for example, in a long `if-else` chain. You'll see it used often.

For

The Go `for` loop is similar to—but not the same as—C's. It unifies `for` and `while` and there is no `do-while`. There are three forms, only one of which has semicolons.

```
// Like a C for
for init; condition; post { }
```

```
// Like a C while
for condition { }
```

```
// Like a C for(;;)
for { }
```

Short declarations make it easy to declare the index variable right in the loop.

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

If you're looping over an array, slice, string, or map, or reading from a channel, a range clause can manage the loop.

```
for key, value := range oldMap {
    newMap[key] = value
}
```

If you only need the first item in the range (the key or index), drop the second:

```
for key := range m {
    if expired(key) {
        delete(m, key)
    }
}
```

If you only need the second item in the range (the value), use the *blank identifier*, an underscore, to discard the first:

```
sum := 0
```

```
for _, value := range array {
    sum += value
}
```

For strings, the range does more work for you, breaking out individual Unicode characters by parsing the UTF-8. Erroneous encodings consume one byte and produce the replacement rune U+FFFD. The loop

```
for pos, char := range "000Z" {
    fmt.Printf("character %c starts at byte position %d\n", char, po
}
```

prints

```
character 00 starts at byte position 0
character 00 starts at byte position 3
character 0Z starts at byte position 6
```

Finally, Go has no comma operator and ++ and -- are statements not expressions. Thus if you want to run multiple variables in a for you should use parallel assignment.

```
// Reverse a
for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
    a[i], a[j] = a[j], a[i]
}
```

Switch

Go's switch is more general than C's. The expressions need not be constants or even integers, the cases are evaluated top to bottom until a match is found, and if the switch has no expression it switches on true. It's therefore possible—and idiomatic—to write an if-else-if-else chain as a switch.

```
func unhex(c byte) byte {
    switch {
    case '0' <= c && c <= '9':
        return c - '0'
    case 'a' <= c && c <= 'f':
        return c - 'a' + 10
    case 'A' <= c && c <= 'F':
        return c - 'A' + 10
    }
    return 0
}
```

There is no automatic fall through, but cases can be presented in comma-separated lists.

```
func shouldEscape(c byte) bool {
    switch c {
    case ' ', '?', '&', '=', '#', '+', '%':
        return true
    }
    return false
}
```

Here's a comparison routine for byte arrays that uses two switch statements:

```
// Compare returns an integer comparing the two byte arrays,
// lexicographically.
// The result will be 0 if a == b, -1 if a < b, and +1 if a > b
func Compare(a, b []byte) int {
    for i := 0; i < len(a) && i < len(b); i++ {
        switch {
        case a[i] > b[i]:
            return 1
        case a[i] < b[i]:
            return -1
        }
    }
    switch {
    case len(a) < len(b):
        return -1
    case len(a) > len(b):
        return 1
    }
    return 0
}
```

A switch can also be used to discover the dynamic type of an interface variable. Such a *type switch* uses the syntax of a type assertion with the keyword `type` inside the parentheses. If the switch declares a variable in the expression, the variable will have the corresponding type in each clause.

```
switch t := interfaceValue.(type) {
default:
    fmt.Printf("unexpected type %T", t) // %T prints type
case bool:
    fmt.Printf("boolean %t\n", t)
case int:
    fmt.Printf("integer %d\n", t)
case *bool:
```

```
    fmt.Printf("pointer to boolean %t\n", *t)
case *int:
    fmt.Printf("pointer to integer %d\n", *t)
}
```

Functions

Multiple return values

One of Go's unusual features is that functions and methods can return multiple values. This form can be used to improve on a couple of clumsy idioms in C programs: in-band error returns (such as -1 for EOF) and modifying an argument.

In C, a write error is signaled by a negative count with the error code secreted away in a volatile location. In Go, `write` can return a count *and* an error: “Yes, you wrote some bytes but not all of them because you filled the device”. The signature of `File.Write` in package `os` is:

```
func (file *File) Write(b []byte) (n int, err error)
```

and as the documentation says, it returns the number of bytes written and a non-nil error when `n != len(b)`. This is a common style; see the section on error handling for more examples.

A similar approach obviates the need to pass a pointer to a return value to simulate a reference parameter. Here's a simple-minded function to grab a number from a position in a byte array, returning the number and the next position.

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {
    }
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x*10 + int(b[i])-'0'
    }
    return x, i
}
```

You could use it to scan the numbers in an input array a like this:

```
for i := 0; i < len(a); {
    x, i = nextInt(a, i)
    fmt.Println(x)
}
```

Named result parameters

The return or result "parameters" of a Go function can be given names and used as regular variables, just like the incoming parameters. When named, they are initialized to the zero values for their types when the function begins; if the function executes a return statement with no arguments, the current values of the result parameters are used as the returned values.

The names are not mandatory but they can make code shorter and clearer: they're documentation. If we name the results of `nextInt` it becomes obvious which returned int is which.

```
func nextInt(b []byte, pos int) (value, nextPos int) {
```

Because named results are initialized and tied to an unadorned return, they can simplify as well as clarify. Here's a version of `io.ReadFull` that uses them well:

```
func ReadFull(r Reader, buf []byte) (n int, err error) {
    for len(buf) > 0 && err == nil {
        var nr int
        nr, err = r.Read(buf)
        n += nr
        buf = buf[nr:]
    }
    return
}
```

Defer

Go's `defer` statement schedules a function call (the *deferred* function) to be run immediately before the function executing the `defer` returns. It's an unusual but effective way to deal with situations such as resources that must be released regardless of which path a function takes to return. The canonical examples are unlocking a mutex or closing a file.

```
// Contents returns the file's contents as a string.
func Contents(filename string) (string, error) {
    f, err := os.Open(filename)
    if err != nil {
        return "", err
    }
    defer f.Close() // f.Close will run when we're finished.
```

```

var result []byte
buf := make([]byte, 100)
for {
    n, err := f.Read(buf[0:])
    result = append(result, buf[0:n]...) // append is discussed
    if err != nil {
        if err == io.EOF {
            break
        }
        return "", err // f will be closed if we return here.
    }
}
return string(result), nil // f will be closed if we return here
}

```

Deferring a call to a function such as `close` has two advantages. First, it guarantees that you will never forget to close the file, a mistake that's easy to make if you later edit the function to add a new return path. Second, it means that the `close` sits near the `open`, which is much clearer than placing it at the end of the function.

The arguments to the deferred function (which include the receiver if the function is a method) are evaluated when the *defer* executes, not when the *call* executes. Besides avoiding worries about variables changing values as the function executes, this means that a single deferred call site can defer multiple function executions. Here's a silly example.

```

for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
}

```

Deferred functions are executed in LIFO order, so this code will cause 4 3 2 1 0 to be printed when the function returns. A more plausible example is a simple way to trace function execution through the program. We could write a couple of simple tracing routines like this:

```

func trace(s string) { fmt.Println("entering:", s) }
func untrace(s string) { fmt.Println("leaving:", s) }

// Use them like this:
func a() {
    trace("a")
    defer untrace("a")
    // do something....
}

```

We can do better by exploiting the fact that arguments to deferred functions are evaluated when the defer executes. The tracing routine can set up the argument to the untracing routine. This example:

```
func trace(s string) string {
    fmt.Println("entering:", s)
    return s
}

func un(s string) {
    fmt.Println("leaving:", s)
}

func a() {
    defer un(trace("a"))
    fmt.Println("in a")
}

func b() {
    defer un(trace("b"))
    fmt.Println("in b")
    a()
}

func main() {
    b()
}
```

prints

```
entering: b
in b
entering: a
in a
leaving: a
leaving: b
```

For programmers accustomed to block-level resource management from other languages, defer may seem peculiar, but its most interesting and powerful applications come precisely from the fact that it's not block-based but function-based. In the section on panic and recover we'll see another example of its possibilities.

Data

Allocation with `new`

Go has two allocation primitives, the built-in functions `new` and `make`. They do different things and apply to different types, which can be confusing, but the rules are simple. Let's talk about `new` first. It's a built-in function that allocates memory, but unlike its namesakes in some other languages it does not *initialize* the memory, it only *zeros* it. That is, `new(T)` allocates zeroed storage for a new item of type `T` and returns its address, a value of type `*T`. In Go terminology, it returns a pointer to a newly allocated zero value of type `T`.

Since the memory returned by `new` is zeroed, it's helpful to arrange when designing your data structures that the zero value of each type can be used without further initialization. This means a user of the data structure can create one with `new` and get right to work. For example, the documentation for `bytes.Buffer` states that "the zero value for `Buffer` is an empty buffer ready to use." Similarly, `sync.Mutex` does not have an explicit constructor or `Init` method. Instead, the zero value for a `sync.Mutex` is defined to be an unlocked mutex.

The zero-value-is-useful property works transitively. Consider this type declaration.

```
type SyncedBuffer struct {  
    lock    sync.Mutex  
    buffer  bytes.Buffer  
}
```

Values of type `SyncedBuffer` are also ready to use immediately upon allocation or just declaration. In the next snippet, both `p` and `v` will work correctly without further arrangement.

```
p := new(SyncedBuffer) // type *SyncedBuffer  
var v SyncedBuffer     // type SyncedBuffer
```

Constructors and composite literals

Sometimes the zero value isn't good enough and an initializing constructor is

necessary, as in this example derived from package `os`.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File)
    f.fd = fd
    f.name = name
    f.dirinfo = nil
    f.nepipe = 0
    return f
}
```

There's a lot of boiler plate in there. We can simplify it using a *composite literal*, which is an expression that creates a new instance each time it is evaluated.

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0}
    return &f
}
```

Note that, unlike in C, it's perfectly OK to return the address of a local variable; the storage associated with the variable survives after the function returns. In fact, taking the address of a composite literal allocates a fresh instance each time it is evaluated, so we can combine these last two lines.

```
    return &File{fd, name, nil, 0}
```

The fields of a composite literal are laid out in order and must all be present. However, by labeling the elements explicitly as *field:value* pairs, the initializers can appear in any order, with the missing ones left as their respective zero values. Thus we could say

```
    return &File{fd: fd, name: name}
```

As a limiting case, if a composite literal contains no fields at all, it creates a zero value for the type. The expressions `new(File)` and `&File{}` are equivalent.

Composite literals can also be created for arrays, slices, and maps, with the field labels being indices or map keys as appropriate. In these examples, the

initializations work regardless of the values of `Enone`, `Eio`, and `Einval`, as long as they are distinct.

```
a := [...]string {Enone: "no error", Eio: "Eio", Einval: "invalid"}
s := []string     {Enone: "no error", Eio: "Eio", Einval: "invalid"}
m := map[int]string{Enone: "no error", Eio: "Eio", Einval: "invalid"}
```

Allocation with `make`

Back to allocation. The built-in function `make(τ , args)` serves a purpose different from `new(τ)`. It creates slices, maps, and channels only, and it returns an *initialized* (not *zeroed*) value of type τ (not $*\tau$). The reason for the distinction is that these three types are, under the covers, references to data structures that must be initialized before use. A slice, for example, is a three-item descriptor containing a pointer to the data (inside an array), the length, and the capacity, and until those items are initialized, the slice is `nil`. For slices, maps, and channels, `make` initializes the internal data structure and prepares the value for use. For instance,

```
make([]int, 10, 100)
```

allocates an array of 100 ints and then creates a slice structure with length 10 and a capacity of 100 pointing at the first 10 elements of the array. (When making a slice, the capacity can be omitted; see the section on slices for more information.) In contrast, `new([]int)` returns a pointer to a newly allocated, zeroed slice structure, that is, a pointer to a `nil` slice value.

These examples illustrate the difference between `new` and `make`.

```
var p *[]int = new([]int) // allocates slice structure; *p == nil
var v []int = make([]int, 100) // the slice v now refers to a new array

// Unnecessarily complex:
var p *[]int = new([]int)
*p = make([]int, 100, 100)

// Idiomatic:
v := make([]int, 100)
```

Remember that `make` applies only to maps, slices and channels and does not return a pointer. To obtain an explicit pointer allocate with `new`.

Arrays

Arrays are useful when planning the detailed layout of memory and sometimes can help avoid allocation, but primarily they are a building block for slices, the subject of the next section. To lay the foundation for that topic, here are a few words about arrays.

There are major differences between the ways arrays work in Go and C. In Go,

- Arrays are values. Assigning one array to another copies all the elements.
- In particular, if you pass an array to a function, it will receive a *copy* of the array, not a pointer to it.
- The size of an array is part of its type. The types `[10]int` and `[20]int` are distinct.

The value property can be useful but also expensive; if you want C-like behavior and efficiency, you can pass a pointer to the array.

```
func Sum(a *[3]float64) (sum float64) {
    for _, v := range *a {
        sum += v
    }
    return
}

array := [...]float64{7.0, 8.5, 9.1}
x := Sum(&array) // Note the explicit address-of operator
```

But even this style isn't idiomatic Go. Slices are.

Slices

Slices wrap arrays to give a more general, powerful, and convenient interface to sequences of data. Except for items with explicit dimension such as transformation matrices, most array programming in Go is done with slices rather than simple arrays.

Slices are *reference types*, which means that if you assign one slice to another, both refer to the same underlying array. For instance, if a function takes a slice argument, changes it makes to the elements of the slice will be visible to the caller, analogous to passing a pointer to the underlying array. A Read function

can therefore accept a slice argument rather than a pointer and a count; the length within the slice sets an upper limit of how much data to read. Here is the signature of the Read method of the File type in package os:

```
func (file *File) Read(buf []byte) (n int, err error)
```

The method returns the number of bytes read and an error value, if any. To read into the first 32 bytes of a larger buffer *b*, *slice* (here used as a verb) the buffer.

```
n, err := f.Read(buf[0:32])
```

Such slicing is common and efficient. In fact, leaving efficiency aside for the moment, the following snippet would also read the first 32 bytes of the buffer.

```
var n int
var err error
for i := 0; i < 32; i++ {
    nbytes, e := f.Read(buf[i:i+1]) // Read one byte.
    if nbytes == 0 || e != nil {
        err = e
        break
    }
    n += nbytes
}
```

The length of a slice may be changed as long as it still fits within the limits of the underlying array; just assign it to a slice of itself. The *capacity* of a slice, accessible by the built-in function *cap*, reports the maximum length the slice may assume. Here is a function to append data to a slice. If the data exceeds the capacity, the slice is reallocated. The resulting slice is returned. The function uses the fact that *len* and *cap* are legal when applied to the *nil* slice, and return 0.

```
func Append(slice, data[]byte) []byte {
    l := len(slice)
    if l + len(data) > cap(slice) { // reallocate
        // Allocate double what's needed, for future growth.
        newSlice := make([]byte, (l+len(data))*2)
        // The copy function is predeclared and works for any slice
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:l+len(data)]
    for i, c := range data {
        slice[l+i] = c
    }
}
```

```
    }
    return slice
}
```

We must return the slice afterwards because, although `Append` can modify the elements of `slice`, the slice itself (the run-time data structure holding the pointer, length, and capacity) is passed by value.

The idea of appending to a slice is so useful it's captured by the `append` built-in function. To understand that function's design, though, we need a little more information, so we'll return to it later.

Maps

Maps are a convenient and powerful built-in data structure to associate values of different types. The key can be of any type for which the equality operator is defined, such as integers, floating point and complex numbers, strings, pointers, interfaces (as long as the dynamic type supports equality), structs and arrays. Slices cannot be used as map keys, because equality is not defined on them. Like slices, maps are a reference type. If you pass a map to a function that changes the contents of the map, the changes will be visible in the caller.

Maps can be constructed using the usual composite literal syntax with colon-separated key-value pairs, so it's easy to build them during initialization.

```
var timeZone = map[string] int {
    "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
}
```

Assigning and fetching map values looks syntactically just like doing the same for arrays except that the index doesn't need to be an integer.

```
offset := timeZone["EST"]
```

An attempt to fetch a map value with a key that is not present in the map will return the zero value for the type of the entries in the map. For instance, if the map contains integers, looking up a non-existent key will return `0`. A set can be implemented as a map with value type `bool`. Set the map entry to `true` to put the

value in the set, and then test it by simple indexing.

```
attended := map[string] bool {
    "Ann": true,
    "Joe": true,
    ...
}

if attended[person] { // will be false if person is not in the map
    fmt.Println(person, "was at the meeting")
}
```

Sometimes you need to distinguish a missing entry from a zero value. Is there an entry for "UTC" or is that zero value because it's not in the map at all? You can discriminate with a form of multiple assignment.

```
var seconds int
var ok bool
seconds, ok = timeZone[tz]
```

For obvious reasons this is called the “comma ok” idiom. In this example, if `tz` is present, `seconds` will be set appropriately and `ok` will be true; if not, `seconds` will be set to zero and `ok` will be false. Here's a function that puts it together with a nice error report:

```
func offset(tz string) int {
    if seconds, ok := timeZone[tz]; ok {
        return seconds
    }
    log.Println("unknown time zone:", tz)
    return 0
}
```

To test for presence in the map without worrying about the actual value, you can use the blank identifier (`_`). The blank identifier can be assigned or declared with any value of any type, with the value discarded harmlessly. For testing just presence in a map, use the blank identifier in place of the usual variable for the value.

```
_, present := timeZone[tz]
```

To delete a map entry, use the `delete` built-in function, whose arguments are the map and the key to be deleted. It's safe to do this even if the key is already absent from the map.

```
delete(timeZone, "PDT") // Now on Standard Time
```

Printing

Formatted printing in Go uses a style similar to C's `printf` family but is richer and more general. The functions live in the `fmt` package and have capitalized names: `fmt.Printf`, `fmt.Fprintf`, `fmt.Sprintf` and so on. The string functions (`Sprintf` etc.) return a string rather than filling in a provided buffer.

You don't need to provide a format string. For each of `Printf`, `Fprintf` and `Sprintf` there is another pair of functions, for instance `Print` and `Println`. These functions do not take a format string but instead generate a default format for each argument. The `Println` versions also insert a blank between arguments and append a newline to the output while the `Print` versions add blanks only if the operand on neither side is a string. In this example each line produces the same output.

```
fmt.Printf("Hello %d\n", 23)
fmt.Fprint(os.Stdout, "Hello ", 23, "\n")
fmt.Println("Hello", 23)
fmt.Println(fmt.Sprint("Hello ", 23))
```

As mentioned in the [Tour](#), `fmt.Fprint` and friends take as a first argument any object that implements the `io.Writer` interface; the variables `os.Stdout` and `os.Stderr` are familiar instances.

Here things start to diverge from C. First, the numeric formats such as `%d` do not take flags for signedness or size; instead, the printing routines use the type of the argument to decide these properties.

```
var x uint64 = 1<<64 - 1
fmt.Printf("%d %x; %d %x\n", x, x, int64(x), int64(x))
```

prints

```
18446744073709551615 ffffffffffffffffffff; -1 -1
```

If you just want the default conversion, such as decimal for integers, you can use the catchall format `%v` (for “value”); the result is exactly what `Print` and `Println` would produce. Moreover, that format can print *any* value, even arrays, structs, and maps. Here is a print statement for the time zone map defined in the

previous section.

```
fmt.Printf("%v\n", timeZone) // or just fmt.Println(timeZone)
```

which gives output

```
map[CST:-21600 PST:-28800 EST:-18000 UTC:0 MST:-25200]
```

For maps the keys may be output in any order, of course. When printing a struct, the modified format `%+v` annotates the fields of the structure with their names, and for any value the alternate format `%#v` prints the value in full Go syntax.

```
type T struct {
    a int
    b float64
    c string
}
t := &T{ 7, -2.35, "abc\tdef" }
fmt.Printf("%v\n", t)
fmt.Printf("%+v\n", t)
fmt.Printf("%#v\n", t)
fmt.Printf("%#v\n", timeZone)
```

prints

```
&{7 -2.35 abc def}
&{a:7 b:-2.35 c:abc def}
&main.T{a:7, b:-2.35, c:"abc\tdef"}
map[string] int{"CST":-21600, "PST":-28800, "EST":-18000, "UTC":0, "
```

(Note the ampersands.) That quoted string format is also available through `%q` when applied to a value of type `string` or `[]byte`; the alternate format `%#q` will use backquotes instead if possible. Also, `%x` works on strings and arrays of bytes as well as on integers, generating a long hexadecimal string, and with a space in the format (`% x`) it puts spaces between the bytes.

Another handy format is `%T`, which prints the *type* of a value.

```
fmt.Printf("%T\n", timeZone)
```

prints

```
map[string] int
```

If you want to control the default format for a custom type, all that's required is to define a method with the signature `String() string` on the type. For our simple type `T`, that might look like this.

```
func (t *T) String() string {
    return fmt.Sprintf("%d/%g/%q", t.a, t.b, t.c)
}
fmt.Printf("%v\n", t)
```

to print in the format

```
7/-2.35/"abc\tdef"
```

(If you need to print *values* of type `T` as well as pointers to `T`, the receiver for `String` must be of value type; this example used a pointer because that's more efficient and idiomatic for struct types. See the section below on [pointers vs. value receivers](#) for more information.)

Our `String` method is able to call `Sprintf` because the print routines are fully reentrant and can be used recursively. We can even go one step further and pass a print routine's arguments directly to another such routine. The signature of `Printf` uses the type `...interface{}` for its final argument to specify that an arbitrary number of parameters (of arbitrary type) can appear after the format.

```
func Printf(format string, v ...interface{}) (n int, err error) {
```

Within the function `Printf`, `v` acts like a variable of type `[]interface{}` but if it is passed to another variadic function, it acts like a regular list of arguments. Here is the implementation of the function `log.Println` we used above. It passes its arguments directly to `fmt.Println` for the actual formatting.

```
//.Println prints to the standard logger in the manner of fmt.Println
func Println(v ...interface{}) {
    std.Output(2, fmt.Println(v...)) // Output takes parameters (i
}
```

We write `...` after `v` in the nested call to `Sprintln` to tell the compiler to treat `v` as a list of arguments; otherwise it would just pass `v` as a single slice argument.

There's even more to printing than we've covered here. See the [godoc documentation](#) for package `fmt` for the details.

By the way, a ... parameter can be of a specific type, for instance ...int for a min function that chooses the least of a list of integers:

```
func Min(a ...int) int {
    min := int(^uint(0) >> 1) // largest int
    for _, i := range a {
        if i < min {
            min = i
        }
    }
    return min
}
```

Append

Now we have the missing piece we needed to explain the design of the append built-in function. The signature of append is different from our custom Append function above. Schematically, it's like this:

```
func append(slice []T, elements...T) []T
```

where T is a placeholder for any given type. You can't actually write a function in Go where the type T is determined by the caller. That's why append is built in: it needs support from the compiler.

What append does is append the elements to the end of the slice and return the result. The result needs to be returned because, as with our hand-written Append, the underlying array may change. This simple example

```
x := []int{1,2,3}
x = append(x, 4, 5, 6)
fmt.Println(x)
```

prints [1 2 3 4 5 6]. So append works a little like Printf, collecting an arbitrary number of arguments.

But what if we wanted to do what our Append does and append a slice to a slice? Easy: use ... at the call site, just as we did in the call to output above. This snippet produces identical output to the one above.

```
x := []int{1,2,3}
y := []int{4,5,6}
x = append(x, y...)
```

```
fmt.Println(x)
```

Without that . . ., it wouldn't compile because the types would be wrong; y is not of type int.

Initialization

Although it doesn't look superficially very different from initialization in C or C++, initialization in Go is more powerful. Complex structures can be built during initialization and the ordering issues between initialized objects in different packages are handled correctly.

Constants

Constants in Go are just that—constant. They are created at compile time, even when defined as locals in functions, and can only be numbers, strings or booleans. Because of the compile-time restriction, the expressions that define them must be constant expressions, evaluatable by the compiler. For instance, `1<<3` is a constant expression, while `math.Sin(math.Pi/4)` is not because the function call to `math.Sin` needs to happen at run time.

In Go, enumerated constants are created using the `iota` enumerator. Since `iota` can be part of an expression and expressions can be implicitly repeated, it is easy to build intricate sets of values.

```
type ByteSize float64

const (
    _ = iota // ignore first value by assigning to blank i
    KB ByteSize = 1 << (10 * iota)
    MB
    GB
    TB
    PB
    EB
    ZB
    YB
)
```

The ability to attach a method such as `String` to a type makes it possible for such values to format themselves automatically for printing, even as part of a general type.

```
func (b ByteSize) String() string {
    switch {
    case b >= YB:
```

```

        return fmt.Sprintf("%.2fYB", b/YB)
    case b >= ZB:
        return fmt.Sprintf("%.2fZB", b/ZB)
    case b >= EB:
        return fmt.Sprintf("%.2fEB", b/EB)
    case b >= PB:
        return fmt.Sprintf("%.2fPB", b/PB)
    case b >= TB:
        return fmt.Sprintf("%.2fTB", b/TB)
    case b >= GB:
        return fmt.Sprintf("%.2fGB", b/GB)
    case b >= MB:
        return fmt.Sprintf("%.2fMB", b/MB)
    case b >= KB:
        return fmt.Sprintf("%.2fKB", b/KB)
    }
    return fmt.Sprintf("%.2fB", b)
}

```

The expression `YB` prints as `1.00YB`, while `ByteSize(1e13)` prints as `9.09TB`.

Note that it's fine to call `Printf` and friends in the implementation of `String` methods, but beware of recurring into the `String` method through the nested `Printf` call using a string format (`%s`, `%q`, `%v`, `%x` or `%X`). The `ByteSize` implementation of `String` is safe because it calls `Printf` with `%f`.

Variables

Variables can be initialized just like constants but the initializer can be a general expression computed at run time.

```

var (
    HOME = os.Getenv("HOME")
    USER = os.Getenv("USER")
    GOROOT = os.Getenv("GOROOT")
)

```

The `init` function

Finally, each source file can define its own `init` function to set up whatever state is required. (Actually each file can have multiple `init` functions.) And finally means finally: `init` is called after all the variable declarations in the package have evaluated their initializers, and those are evaluated only after all the imported packages have been initialized.

Besides initializations that cannot be expressed as declarations, a common use of init functions is to verify or repair correctness of the program state before real execution begins.

```
func init() {
    if USER == "" {
        log.Fatal("$USER not set")
    }
    if HOME == "" {
        HOME = "/usr/" + USER
    }
    if GOROOT == "" {
        GOROOT = HOME + "/go"
    }
    // GOROOT may be overridden by --goroot flag on command line.
    flag.StringVar(&GOROOT, "goroot", GOROOT, "Go root directory")
}
```

Methods

Pointers vs. Values

Methods can be defined for any named type that is not a pointer or an interface; the receiver does not have to be a struct.

In the discussion of slices above, we wrote an `Append` function. We can define it as a method on slices instead. To do this, we first declare a named type to which we can bind the method, and then make the receiver for the method a value of that type.

```
type ByteSlice []byte

func (slice ByteSlice) Append(data []byte) []byte {
    // Body exactly the same as above
}
```

This still requires the method to return the updated slice. We can eliminate that clumsiness by redefining the method to take a *pointer* to a `ByteSlice` as its receiver, so the method can overwrite the caller's slice.

```
func (p *ByteSlice) Append(data []byte) {
    slice := *p
    // Body as above, without the return.
    *p = slice
}
```

In fact, we can do even better. If we modify our function so it looks like a standard write method, like this,

```
func (p *ByteSlice) Write(data []byte) (n int, err error) {
    slice := *p
    // Again as above.
    *p = slice
    return len(data), nil
}
```

then the type `*ByteSlice` satisfies the standard interface `io.Writer`, which is handy. For instance, we can print into one.

```
var b ByteSlice
```

```
fmt.Fprintf(&b, "This hour has %d days\n", 7)
```

We pass the address of a `ByteSlice` because only `*ByteSlice` satisfies `io.Writer`. The rule about pointers vs. values for receivers is that value methods can be invoked on pointers and values, but pointer methods can only be invoked on pointers. This is because pointer methods can modify the receiver; invoking them on a copy of the value would cause those modifications to be discarded.

By the way, the idea of using `write` on a slice of bytes is implemented by `bytes.Buffer`.

Interfaces and other types

Interfaces

Interfaces in Go provide a way to specify the behavior of an object: if something can do *this*, then it can be used *here*. We've seen a couple of simple examples already; custom printers can be implemented by a `String` method while `Fprintf` can generate output to anything with a `Write` method. Interfaces with only one or two methods are common in Go code, and are usually given a name derived from the method, such as `io.Writer` for something that implements `Write`.

A type can implement multiple interfaces. For instance, a collection can be sorted by the routines in package `sort` if it implements `sort.Interface`, which contains `Len()`, `Less(i, j int) bool`, and `Swap(i, j int)`, and it could also have a custom formatter. In this contrived example `Sequence` satisfies both.

```
type Sequence []int

// Methods required by sort.Interface.
func (s Sequence) Len() int {
    return len(s)
}
func (s Sequence) Less(i, j int) bool {
    return s[i] < s[j]
}
func (s Sequence) Swap(i, j int) {
    s[i], s[j] = s[j], s[i]
}

// Method for printing - sorts the elements before printing.
func (s Sequence) String() string {
    sort.Sort(s)
    str := "["
    for i, elem := range s {
        if i > 0 {
            str += " "
        }
        str += fmt.Sprintf(elem)
    }
    return str + "]"
}
```

Conversions

The `String` method of `Sequence` is recreating the work that `Sprint` already does for slices. We can share the effort if we convert the `Sequence` to a plain `[]int` before calling `Sprint`.

```
func (s Sequence) String() string {
    sort.Sort(s)
    return fmt.Sprint([]int(s))
}
```

The conversion causes `s` to be treated as an ordinary slice and therefore receive the default formatting. Without the conversion, `Sprint` would find the `String` method of `Sequence` and recur indefinitely. Because the two types (`Sequence` and `[]int`) are the same if we ignore the type name, it's legal to convert between them. The conversion doesn't create a new value, it just temporarily acts as though the existing value has a new type. (There are other legal conversions, such as from integer to floating point, that do create a new value.)

It's an idiom in Go programs to convert the type of an expression to access a different set of methods. As an example, we could use the existing type `sort.IntSlice` to reduce the entire example to this:

```
type Sequence []int

// Method for printing - sorts the elements before printing
func (s Sequence) String() string {
    sort.IntSlice(s).Sort()
    return fmt.Sprint([]int(s))
}
```

Now, instead of having `Sequence` implement multiple interfaces (sorting and printing), we're using the ability of a data item to be converted to multiple types (`Sequence`, `sort.IntSlice` and `[]int`), each of which does some part of the job. That's more unusual in practice but can be effective.

Generality

If a type exists only to implement an interface and has no exported methods beyond that interface, there is no need to export the type itself. Exporting just the interface makes it clear that it's the behavior that matters, not the implementation, and that other implementations with different properties can mirror the behavior of the original type. It also avoids the need to repeat the documentation on every instance of a common method.

In such cases, the constructor should return an interface value rather than the implementing type. As an example, in the hash libraries both `crc32.NewIEEE` and `adler32.New` return the interface type `hash.Hash32`. Substituting the CRC-32 algorithm for Adler-32 in a Go program requires only changing the constructor call; the rest of the code is unaffected by the change of algorithm.

A similar approach allows the streaming cipher algorithms in the various `crypto` packages to be separated from the block ciphers they chain together. The `Block` interface in the `crypto/cipher` package specifies the behavior of a block cipher, which provides encryption of a single block of data. Then, by analogy with the `bufio` package, cipher packages that implement this interface can be used to construct streaming ciphers, represented by the `Stream` interface, without knowing the details of the block encryption.

The `crypto/cipher` interfaces look like this:

```
type Block interface {
    BlockSize() int
    Encrypt(src, dst []byte)
    Decrypt(src, dst []byte)
}

type Stream interface {
    XORKeyStream(dst, src []byte)
}
```

Here's the definition of the counter mode (CTR) stream, which turns a block cipher into a streaming cipher; notice that the block cipher's details are abstracted away:

```
// NewCTR returns a Stream that encrypts/decrypts using the given Block
// counter mode. The length of iv must be the same as the Block's block
func NewCTR(block Block, iv []byte) Stream
```

`NewCTR` applies not just to one specific encryption algorithm and data source but to any implementation of the `Block` interface and any `Stream`. Because they return interface values, replacing CTR encryption with other encryption modes is a localized change. The constructor calls must be edited, but because the surrounding code must treat the result only as a `Stream`, it won't notice the difference.

Interfaces and methods

Since almost anything can have methods attached, almost anything can satisfy an interface. One illustrative example is in the `http` package, which defines the `Handler` interface. Any object that implements `Handler` can serve HTTP requests.

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

`ResponseWriter` is itself an interface that provides access to the methods needed to return the response to the client. Those methods include the standard `write` method, so an `http.ResponseWriter` can be used wherever an `io.Writer` can be used. `Request` is a struct containing a parsed representation of the request from the client.

For brevity, let's ignore POSTs and assume HTTP requests are always GETs; that simplification does not affect the way the handlers are set up. Here's a trivial but complete implementation of a handler to count the number of times the page is visited.

```
// Simple counter server.
type Counter struct {
    n int
}

func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Reque
    ctr.n++
    fmt.Fprintf(w, "counter = %d\n", ctr.n)
}
```

(Keeping with our theme, note how `Fprintf` can print to an `http.ResponseWriter`.) For reference, here's how to attach such a server to a node on the URL tree.

```
import "net/http"
...
ctr := new(Counter)
http.Handle("/counter", ctr)
```

But why make `Counter` a struct? An integer is all that's needed. (The receiver needs to be a pointer so the increment is visible to the caller.)

```
// Simpler counter server.
type Counter int
```

```
func (ctr *Counter) ServeHTTP(w http.ResponseWriter, req *http.Reque
    *ctr++
    fmt.Fprintf(w, "counter = %d\n", *ctr)
}
```

What if your program has some internal state that needs to be notified that a page has been visited? Tie a channel to the web page.

```
// A channel that sends a notification on each visit.
// (Probably want the channel to be buffered.)
type Chan chan *http.Request

func (ch Chan) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    ch <- req
    fmt.Fprint(w, "notification sent")
}
```

Finally, let's say we wanted to present on `/args` the arguments used when invoking the server binary. It's easy to write a function to print the arguments.

```
func ArgServer() {
    for _, s := range os.Args {
        fmt.Println(s)
    }
}
```

How do we turn that into an HTTP server? We could make `ArgServer` a method of some type whose value we ignore, but there's a cleaner way. Since we can define a method for any type except pointers and interfaces, we can write a method for a function. The `http` package contains this code:

```
// The HandlerFunc type is an adapter to allow the use of
// ordinary functions as HTTP handlers. If f is a function
// with the appropriate signature, HandlerFunc(f) is a
// Handler object that calls f.
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(c, req).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, req *Request) {
    f(w, req)
}
```

`HandlerFunc` is a type with a method, `ServeHTTP`, so values of that type can serve HTTP requests. Look at the implementation of the method: the receiver is a function, `f`, and the method calls `f`. That may seem odd but it's not that

different from, say, the receiver being a channel and the method sending on the channel.

To make `ArgServer` into an HTTP server, we first modify it to have the right signature.

```
// Argument server.  
func ArgServer(w http.ResponseWriter, req *http.Request) {  
    for _, s := range os.Args {  
        fmt.Fprintln(w, s)  
    }  
}
```

`ArgServer` now has same signature as `HandlerFunc`, so it can be converted to that type to access its methods, just as we converted `Sequence` to `IntSlice` to access `IntSlice.Sort`. The code to set it up is concise:

```
http.Handle("/args", http.HandlerFunc(ArgServer))
```

When someone visits the page `/args`, the handler installed at that page has value `ArgServer` and type `HandlerFunc`. The HTTP server will invoke the method `ServeHTTP` of that type, with `ArgServer` as the receiver, which will in turn call `ArgServer` (via the invocation `f(c, req)` inside `HandlerFunc.ServeHTTP`). The arguments will then be displayed.

In this section we have made an HTTP server from a struct, an integer, a channel, and a function, all because interfaces are just sets of methods, which can be defined for (almost) any type.

Embedding

Go does not provide the typical, type-driven notion of subclassing, but it does have the ability to “borrow” pieces of an implementation by *embedding* types within a struct or interface.

Interface embedding is very simple. We've mentioned the `io.Reader` and `io.Writer` interfaces before; here are their definitions.

```
type Reader interface {
    Read(p []byte) (n int, err error)
}

type Writer interface {
    Write(p []byte) (n int, err error)
}
```

The `io` package also exports several other interfaces that specify objects that can implement several such methods. For instance, there is `io.ReadWriter`, an interface containing both `Read` and `write`. We could specify `io.ReadWriter` by listing the two methods explicitly, but it's easier and more evocative to embed the two interfaces to form the new one, like this:

```
// ReadWriter is the interface that combines the Reader and Writer i
type ReadWriter interface {
    Reader
    Writer
}
```

This says just what it looks like: A `ReadWriter` can do what a `Reader` does *and* what a `writer` does; it is a union of the embedded interfaces (which must be disjoint sets of methods). Only interfaces can be embedded within interfaces.

The same basic idea applies to structs, but with more far-reaching implications. The `bufio` package has two struct types, `bufio.Reader` and `bufio.Writer`, each of which of course implements the analogous interfaces from package `io`. And `bufio` also implements a buffered reader/writer, which it does by combining a reader and a writer into one struct using embedding: it lists the types within the struct but does not give them field names.

```
// ReadWriter stores pointers to a Reader and a Writer.
```

```
// It implements io.ReadWriter.
type ReadWriter struct {
    *Reader // *bufio.Reader
    *Writer // *bufio.Writer
}
```

The embedded elements are pointers to structs and of course must be initialized to point to valid structs before they can be used. The `ReadWriter` struct could be written as

```
type ReadWriter struct {
    reader *Reader
    writer *Writer
}
```

but then to promote the methods of the fields and to satisfy the `io` interfaces, we would also need to provide forwarding methods, like this:

```
func (rw *ReadWriter) Read(p []byte) (n int, err error) {
    return rw.reader.Read(p)
}
```

By embedding the structs directly, we avoid this bookkeeping. The methods of embedded types come along for free, which means that `bufio.ReadWriter` not only has the methods of `bufio.Reader` and `bufio.Writer`, it also satisfies all three interfaces: `io.Reader`, `io.Writer`, and `io.ReadWriter`.

There's an important way in which embedding differs from subclassing. When we embed a type, the methods of that type become methods of the outer type, but when they are invoked the receiver of the method is the inner type, not the outer one. In our example, when the `Read` method of a `bufio.ReadWriter` is invoked, it has exactly the same effect as the forwarding method written out above; the receiver is the `reader` field of the `ReadWriter`, not the `ReadWriter` itself.

Embedding can also be a simple convenience. This example shows an embedded field alongside a regular, named field.

```
type Job struct {
    Command string
    *log.Logger
}
```

The Job type now has the Log, Logf and other methods of `*log.Logger`. We could have given the Logger a field name, of course, but it's not necessary to do so. And now, once initialized, we can log to the Job:

```
job.Log("starting now...")
```

The Logger is a regular field of the struct and we can initialize it in the usual way with a constructor,

```
func NewJob(command string, logger *log.Logger) *Job {  
    return &Job{command, logger}  
}
```

or with a composite literal,

```
job := &Job{command, log.New(os.Stderr, "Job: ", log.Ldate)}
```

If we need to refer to an embedded field directly, the type name of the field, ignoring the package qualifier, serves as a field name. If we needed to access the `*log.Logger` of a Job variable `job`, we would write `job.Logger`. This would be useful if we wanted to refine the methods of Logger.

```
func (job *Job) Logf(format string, args ...interface{}) {  
    job.Logger.Logf("%q: %s", job.Command, fmt.Sprintf(format, args...))  
}
```

Embedding types introduces the problem of name conflicts but the rules to resolve them are simple. First, a field or method `x` hides any other item `x` in a more deeply nested part of the type. If `log.Logger` contained a field or method called `Command`, the `Command` field of `Job` would dominate it.

Second, if the same name appears at the same nesting level, it is usually an error; it would be erroneous to embed `log.Logger` if the `Job` struct contained another field or method called `Logger`. However, if the duplicate name is never mentioned in the program outside the type definition, it is OK. This qualification provides some protection against changes made to types embedded from outside; there is no problem if a field is added that conflicts with another field in another subtype if neither field is ever used.

Concurrency

Share by communicating

Concurrent programming is a large topic and there is space only for some Go-specific highlights here.

Concurrent programming in many environments is made difficult by the subtleties required to implement correct access to shared variables. Go encourages a different approach in which shared values are passed around on channels and, in fact, never actively shared by separate threads of execution. Only one goroutine has access to the value at any given time. Data races cannot occur, by design. To encourage this way of thinking we have reduced it to a slogan:

Do not communicate by sharing memory; instead, share memory by communicating.

This approach can be taken too far. Reference counts may be best done by putting a mutex around an integer variable, for instance. But as a high-level approach, using channels to control access makes it easier to write clear, correct programs.

One way to think about this model is to consider a typical single-threaded program running on one CPU. It has no need for synchronization primitives. Now run another such instance; it too needs no synchronization. Now let those two communicate; if the communication is the synchronizer, there's still no need for other synchronization. Unix pipelines, for example, fit this model perfectly. Although Go's approach to concurrency originates in Hoare's Communicating Sequential Processes (CSP), it can also be seen as a type-safe generalization of Unix pipes.

Goroutines

They're called *goroutines* because the existing terms—threads, coroutines, processes, and so on—convey inaccurate connotations. A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same

address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required.

Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and management.

Prefix a function or method call with the `go` keyword to run the call in a new goroutine. When the call completes, the goroutine exits, silently. (The effect is similar to the Unix shell's `&` notation for running a command in the background.)

```
go list.Sort() // run list.Sort concurrently; don't wait for it.
```

A function literal can be handy in a goroutine invocation.

```
func Announce(message string, delay time.Duration) {
    go func() {
        time.Sleep(delay)
        fmt.Println(message)
    }() // Note the parentheses - must call the function.
}
```

In Go, function literals are closures: the implementation makes sure the variables referred to by the function survive as long as they are active.

These examples aren't too practical because the functions have no way of signaling completion. For that, we need channels.

Channels

Like maps, channels are a reference type and are allocated with `make`. If an optional integer parameter is provided, it sets the buffer size for the channel. The default is zero, for an unbuffered or synchronous channel.

```
ci := make(chan int)           // unbuffered channel of integers
cj := make(chan int, 0)       // unbuffered channel of integers
cs := make(chan *os.File, 100) // buffered channel of pointers to F
```

Channels combine communication—the exchange of a value—with synchronization—guaranteeing that two calculations (goroutines) are in a known state.

There are lots of nice idioms using channels. Here's one to get us started. In the previous section we launched a sort in the background. A channel can allow the launching goroutine to wait for the sort to complete.

```
c := make(chan int) // Allocate a channel.
// Start the sort in a goroutine; when it completes, signal on the c
go func() {
    list.Sort()
    c <- 1 // Send a signal; value does not matter.
}()
doSomethingForAWhile()
<-c // Wait for sort to finish; discard sent value.
```

Receivers always block until there is data to receive. If the channel is unbuffered, the sender blocks until the receiver has received the value. If the channel has a buffer, the sender blocks only until the value has been copied to the buffer; if the buffer is full, this means waiting until some receiver has retrieved a value.

A buffered channel can be used like a semaphore, for instance to limit throughput. In this example, incoming requests are passed to `handle`, which sends a value into the channel, processes the request, and then receives a value from the channel. The capacity of the channel buffer limits the number of simultaneous calls to process.

```
var sem = make(chan int, MaxOutstanding)

func handle(r *Request) {
    sem <- 1 // Wait for active queue to drain.
    process(r) // May take a long time.
    <-sem // Done; enable next request to run.
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req) // Don't wait for handle to finish.
    }
}
```

Here's the same idea implemented by starting a fixed number of `handle` goroutines all reading from the request channel. The number of goroutines limits the number of simultaneous calls to process. This `Serve` function also accepts a channel on which it will be told to exit; after launching the goroutines it blocks

receiving from that channel.

```
func handle(queue chan *Request) {
    for r := range queue {
        process(r)
    }
}

func Serve(clientRequests chan *Request, quit chan bool) {
    // Start handlers
    for i := 0; i < MaxOutstanding; i++ {
        go handle(clientRequests)
    }
    <-quit // Wait to be told to exit.
}
```

Channels of channels

One of the most important properties of Go is that a channel is a first-class value that can be allocated and passed around like any other. A common use of this property is to implement safe, parallel demultiplexing.

In the example in the previous section, `handle` was an idealized handler for a request but we didn't define the type it was handling. If that type includes a channel on which to reply, each client can provide its own path for the answer. Here's a schematic definition of type `Request`.

```
type Request struct {
    args      []int
    f         func([]int) int
    resultChan chan int
}
```

The client provides a function and its arguments, as well as a channel inside the request object on which to receive the answer.

```
func sum(a []int) (s int) {
    for _, v := range a {
        s += v
    }
    return
}

request := &Request{[]int{3, 4, 5}, sum, make(chan int)}
// Send request
```

```
clientRequests <- request
// Wait for response.
fmt.Printf("answer: %d\n", <-request.resultChan)
```

On the server side, the handler function is the only thing that changes.

```
func handle(queue chan *Request) {
    for req := range queue {
        req.resultChan <- req.f(req.args)
    }
}
```

There's clearly a lot more to do to make it realistic, but this code is a framework for a rate-limited, parallel, non-blocking RPC system, and there's not a mutex in sight.

Parallelization

Another application of these ideas is to parallelize a calculation across multiple CPU cores. If the calculation can be broken into separate pieces that can execute independently, it can be parallelized, with a channel to signal when each piece completes.

Let's say we have an expensive operation to perform on a vector of items, and that the value of the operation on each item is independent, as in this idealized example.

```
type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to v[n-1].
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1    // signal that this piece is done
}
```

We launch the pieces independently in a loop, one per CPU. They can complete in any order but it doesn't matter; we just count the completion signals by draining the channel after launching all the goroutines.

```
const NCPU = 4 // number of CPU cores

func (v Vector) DoAll(u Vector) {
```

```

    c := make(chan int, NCPU) // Buffering optional but sensible.
    for i := 0; i < NCPU; i++ {
        go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < NCPU; i++ {
        <-c // wait for one task to complete
    }
    // All done.
}

```

The current implementation of the Go runtime will not parallelize this code by default. It dedicates only a single core to user-level processing. An arbitrary number of goroutines can be blocked in system calls, but by default only one can be executing user-level code at any time. It should be smarter and one day it will be smarter, but until it is if you want CPU parallelism you must tell the run-time how many goroutines you want executing code simultaneously. There are two related ways to do this. Either run your job with environment variable `GOMAXPROCS` set to the number of cores to use or import the `runtime` package and call `runtime.GOMAXPROCS(NCPU)`. A helpful value might be `runtime.NumCPU()`, which reports the number of logical CPUs on the local machine. Again, this requirement is expected to be retired as the scheduling and run-time improve.

A leaky buffer

The tools of concurrent programming can even make non-concurrent ideas easier to express. Here's an example abstracted from an RPC package. The client goroutine loops receiving data from some source, perhaps a network. To avoid allocating and freeing buffers, it keeps a free list, and uses a buffered channel to represent it. If the channel is empty, a new buffer gets allocated. Once the message buffer is ready, it's sent to the server on `serverChan`.

```

var freeList = make(chan *Buffer, 100)
var serverChan = make(chan *Buffer)

func client() {
    for {
        var b *Buffer
        // Grab a buffer if available; allocate if not.
        select {
        case b = <-freeList:
            // Got one; nothing more to do.
        default:

```

```

        // None free, so allocate a new one.
        b = new(Buffer)
    }
    load(b)           // Read next message from the net.
    serverChan <- b  // Send to server.
}
}

```

The server loop receives each message from the client, processes it, and returns the buffer to the free list.

```

func server() {
    for {
        b := <-serverChan // Wait for work.
        process(b)
        // Reuse buffer if there's room.
        select {
        case freeList <- b:
            // Buffer on free list; nothing more to do.
        default:
            // Free list full, just carry on.
        }
    }
}

```

The client attempts to retrieve a buffer from `freeList`; if none is available, it allocates a fresh one. The server's send to `freeList` puts `b` back on the free list unless the list is full, in which case the buffer is dropped on the floor to be reclaimed by the garbage collector. (The `default` clauses in the `select` statements execute when no other case is ready, meaning that the selects never block.) This implementation builds a leaky bucket free list in just a few lines, relying on the buffered channel and the garbage collector for bookkeeping.

Errors

Library routines must often return some sort of error indication to the caller. As mentioned earlier, Go's multivalue return makes it easy to return a detailed error description alongside the normal return value. By convention, errors have type `error`, a simple built-in interface.

```
type error interface {
    Error() string
}
```

A library writer is free to implement this interface with a richer model under the covers, making it possible not only to see the error but also to provide some context. For example, `os.Open` returns an `os.PathError`.

```
// PathError records an error and the operation and
// file path that caused it.
type PathError struct {
    Op string    // "open", "unlink", etc.
    Path string  // The associated file.
    Err error      // Returned by the system call.
}

func (e *PathError) Error() string {
    return e.Op + " " + e.Path + ": " + e.Err.Error()
}
```

`PathError`'s `Error` generates a string like this:

```
open /etc/passwx: no such file or directory
```

Such an error, which includes the problematic file name, the operation, and the operating system error it triggered, is useful even if printed far from the call that caused it; it is much more informative than the plain "no such file or directory".

When feasible, error strings should identify their origin, such as by having a prefix naming the package that generated the error. For example, in package `image`, the string representation for a decoding error due to an unknown format is "image: unknown format".

Callers that care about the precise error details can use a type switch or a type

assertion to look for specific errors and extract details. For `PathErrors` this might include examining the internal `Err` field for recoverable failures.

```
for try := 0; try < 2; try++ {
    file, err = os.Create(filename)
    if err == nil {
        return
    }
    if e, ok := err.(*os.PathError); ok && e.Err == syscall.ENOSPC {
        deleteTempFiles() // Recover some space.
        continue
    }
    return
}
```

The second `if` statement here is idiomatic Go. The type assertion `err.(*os.PathError)` is checked with the "comma ok" idiom (mentioned [earlier](#) in the context of examining maps). If the type assertion fails, `ok` will be `false`, and `e` will be `nil`. If it succeeds, `ok` will be `true`, which means the error was of type `*os.PathError`, and then so is `e`, which we can examine for more information about the error.

Panic

The usual way to report an error to a caller is to return an error as an extra return value. The canonical `Read` method is a well-known instance; it returns a byte count and an error. But what if the error is unrecoverable? Sometimes the program simply cannot continue.

For this purpose, there is a built-in function `panic` that in effect creates a run-time error that will stop the program (but see the next section). The function takes a single argument of arbitrary type—often a string—to be printed as the program dies. It's also a way to indicate that something impossible has happened, such as exiting an infinite loop. In fact, the compiler recognizes a panic at the end of a function and suppresses the usual check for a return statement.

```
// A toy implementation of cube root using Newton's method.
func CubeRoot(x float64) float64 {
    z := x/3 // Arbitrary initial value
    for i := 0; i < 1e6; i++ {
        prevz := z
        z -= (z*z*z-x) / (3*z*z)
        if veryClose(z, prevz) {
```

```

        return z
    }
}
// A million iterations has not converged; something is wrong.
panic(fmt.Sprintf("CubeRoot(%g) did not converge", x))
}

```

This is only an example but real library functions should avoid panic. If the problem can be masked or worked around, it's always better to let things continue to run rather than taking down the whole program. One possible counterexample is during initialization: if the library truly cannot set itself up, it might be reasonable to panic, so to speak.

```

var user = os.Getenv("USER")

func init() {
    if user == "" {
        panic("no value for $USER")
    }
}

```

Recover

When panic is called, including implicitly for run-time errors such as indexing an array out of bounds or failing a type assertion, it immediately stops execution of the current function and begins unwinding the stack of the goroutine, running any deferred functions along the way. If that unwinding reaches the top of the goroutine's stack, the program dies. However, it is possible to use the built-in function recover to regain control of the goroutine and resume normal execution.

A call to recover stops the unwinding and returns the argument passed to panic. Because the only code that runs while unwinding is inside deferred functions, recover is only useful inside deferred functions.

One application of recover is to shut down a failing goroutine inside a server without killing the other executing goroutines.

```

func server(workChan <-chan *Work) {
    for work := range workChan {
        go safelyDo(work)
    }
}

```

```

func safelyDo(work *Work) {
    defer func() {
        if err := recover(); err != nil {
            log.Println("work failed:", err)
        }
    }()
    do(work)
}

```

In this example, if `do(work)` panics, the result will be logged and the goroutine will exit cleanly without disturbing the others. There's no need to do anything else in the deferred closure; calling `recover` handles the condition completely.

Because `recover` always returns `nil` unless called directly from a deferred function, deferred code can call library routines that themselves use `panic` and `recover` without failing. As an example, the deferred function in `safelyDo` might call a logging function before calling `recover`, and that logging code would run unaffected by the panicking state.

With our recovery pattern in place, the `do` function (and anything it calls) can get out of any bad situation cleanly by calling `panic`. We can use that idea to simplify error handling in complex software. Let's look at an idealized excerpt from the `regexp` package, which reports parsing errors by calling `panic` with a local error type. Here's the definition of `Error`, an error method, and the `Compile` function.

```

// Error is the type of a parse error; it satisfies the error interf
type Error string
func (e Error) Error() string {
    return string(e)
}

// error is a method of *Regexp that reports parsing errors by
// panicking with an Error.
func (regexp *Regexp) error(err string) {
    panic(Error(err))
}

// Compile returns a parsed representation of the regular expression
func Compile(str string) (regexp *Regexp, err error) {
    regexp = new(Regexp)
    // doParse will panic if there is a parse error.
    defer func() {
        if e := recover(); e != nil {

```

```

        regexp = nil    // Clear return value.
        err = e.(Error) // Will re-panic if not a parse error.
    }
}()
return regexp.doParse(str), nil
}

```

If `doParse` panics, the recovery block will set the return value to `nil`—deferred functions can modify named return values. It then will then check, in the assignment to `err`, that the problem was a parse error by asserting that it has the local type `Error`. If it does not, the type assertion will fail, causing a run-time error that continues the stack unwinding as though nothing had interrupted it. This check means that if something unexpected happens, such as an array index out of bounds, the code will fail even though we are using `panic` and `recover` to handle user-triggered errors.

With error handling in place, the error method makes it easy to report parse errors without worrying about unwinding the parse stack by hand.

Useful though this pattern is, it should be used only within a package. `Parse` turns its internal panic calls into error values; it does not expose panics to its client. That is a good rule to follow.

By the way, this re-panic idiom changes the panic value if an actual error occurs. However, both the original and new failures will be presented in the crash report, so the root cause of the problem will still be visible. Thus this simple re-panic approach is usually sufficient—it's a crash after all—but if you want to display only the original value, you can write a little more code to filter unexpected problems and re-panic with the original error. That's left as an exercise for the reader.

A web server

Let's finish with a complete Go program, a web server. This one is actually a kind of web re-server. Google provides a service at <http://chart.apis.google.com> that does automatic formatting of data into charts and graphs. It's hard to use interactively, though, because you need to put the data into the URL as a query. The program here provides a nicer interface to one form of data: given a short piece of text, it calls on the chart server to produce a QR code, a matrix of boxes that encode the text. That image can be grabbed with your cell phone's camera and interpreted as, for instance, a URL, saving you typing the URL into the phone's tiny keyboard.

Here's the complete program. An explanation follows.

```
package main

import (
    "flag"
    "log"
    "net/http"
    "text/template"
)

var addr = flag.String("addr", ":1718", "http service address") // Q
var templ = template.Must(template.New("qr").Parse(templateStr))

func main() {
    flag.Parse()
    http.Handle("/", http.HandlerFunc(QR))
    err := http.ListenAndServe(*addr, nil)
    if err != nil {
        log.Fatal("ListenAndServe:", err)
    }
}

func QR(w http.ResponseWriter, req *http.Request) {
    templ.Execute(w, req.FormValue("s"))
}

const templateStr = `
<html>
<head>
<title>QR Link Generator</title>
```

```

</head>
<body>
{{if .}}
<input maxLength=1024 size=70
name=s value="" title="Text to QR Encode"><input type=submit
value="Show QR" name=qr>
</form>
</body>
</html>
`

```

The pieces up to main should be easy to follow. The one flag sets a default HTTP port for our server. The template variable `templ` is where the fun happens. It builds an HTML template that will be executed by the server to display the page; more about that in a moment.

The main function parses the flags and, using the mechanism we talked about above, binds the function `QR` to the root path for the server. Then `http.ListenAndServe` is called to start the server; it blocks while the server runs.

`QR` just receives the request, which contains form data, and executes the template on the data in the form value named `s`.

The template package is powerful; this program just touches on its capabilities. In essence, it rewrites a piece of text on the fly by substituting elements derived from data items passed to `templ.Execute`, in this case the form value. Within the template text (`templateStr`), double-brace-delimited pieces denote template actions. The piece from `{{if .}}` to `{{end}}` executes only if the value of the current data item, called `.` (dot), is non-empty. That is, when the string is empty, this piece of the template is suppressed.

The snippet `{{urlquery .}}` says to process the data with the function `urlquery`, which sanitizes the query string for safe display on the web page.

The rest of the template string is just the HTML to show when the page loads. If this is too quick an explanation, see the [documentation](#) for the template package

for a more thorough discussion.

And there you have it: a useful web server in a few lines of code plus some data-driven HTML text. Go is powerful enough to make a lot happen in a few lines.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Command go

Go is a tool for managing Go source code.

Usage:

```
go command [arguments]
```

The commands are:

build	compile packages and dependencies
clean	remove object files
doc	run godoc on package sources
env	print Go environment information
fix	run go tool fix on packages
fmt	run gofmt on package sources
get	download and install packages and dependencies
install	compile and install packages and dependencies
list	list packages
run	compile and run Go program
test	test packages
tool	run specified go tool
version	print Go version
vet	run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

gopath	GOPATH environment variable
packages	description of package lists
remote	remote import path syntax
testflag	description of testing flags
testfunc	description of testing functions

Use "go help [topic]" for more information about that topic.

Compile packages and dependencies

Usage:

```
go build [-o output] [build flags] [packages]
```

Build compiles the packages named by the import paths, along with their dependencies, but it does not install the results.

If the arguments are a list of .go files, build treats them as a list of source files specifying a single package.

When the command line specifies a single main package, build writes the resulting executable to output. Otherwise build compiles the packages but discards the results, serving only as a check that the packages can be built.

The -o flag specifies the output file name. If not specified, the name is packagename.a (for a non-main package) or the base name of the first source file (for a main package).

The build flags are shared by the build, install, run, and test commands:

```
-a          force rebuilding of packages that are already up-to-date.
-n          print the commands but do not run them.
-p n       the number of builds that can be run in parallel.
           The default is the number of CPUs available.
-v          print the names of packages as they are compiled.
-work      print the name of the temporary work directory and
           do not delete it when exiting.
-x          print the commands.

-compiler name
           name of compiler to use, as in runtime.Compiler (gccgo or gc
-gccgoflags 'arg list'
           arguments to pass on each gccgo compiler/linker invocation
-gcflags 'arg list'
           arguments to pass on each 5g, 6g, or 8g compiler invocation
-ldflags 'flag list'
           arguments to pass on each 5l, 6l, or 8l linker invocation
-tags 'tag list'
           a list of build tags to consider satisfied during the build.
           See the documentation for the go/build package for
           more information about build tags.
```

For more about specifying packages, see 'go help packages'. For more about

where packages and binaries are installed, see 'go help gopath'.

See also: go install, go get, go clean.

Remove object files

Usage:

```
go clean [-i] [-r] [-n] [-x] [packages]
```

Clean removes object files from package source directories. The go command builds most objects in a temporary directory, so go clean is mainly concerned with object files left by other tools or by manual invocations of go build.

Specifically, clean removes the following files from each of the source directories corresponding to the import paths:

<code>_obj/</code>	old object directory, left from Makefiles
<code>_test/</code>	old test directory, left from Makefiles
<code>_testmain.go</code>	old gotest file, left from Makefiles
<code>test.out</code>	old test log, left from Makefiles
<code>build.out</code>	old test log, left from Makefiles
<code>*.[568ao]</code>	object files, left from Makefiles
<code>DIR(.exe)</code>	from go build
<code>DIR.test(.exe)</code>	from go test -c
<code>MAINFILE(.exe)</code>	from go build MAINFILE.go

In the list, DIR represents the final path element of the directory, and MAINFILE is the base name of any Go source file in the directory that is not included when building the package.

The -i flag causes clean to remove the corresponding installed archive or binary (what 'go install' would create).

The -n flag causes clean to print the remove commands it would execute, but not run them.

The -r flag causes clean to be applied recursively to all the dependencies of the packages named by the import paths.

The -x flag causes clean to print remove commands as it executes them.

For more about specifying packages, see 'go help packages'.

Run godoc on package sources

Usage:

```
go doc [packages]
```

Doc runs the godoc command on the packages named by the import paths.

For more about godoc, see 'godoc godoc'. For more about specifying packages, see 'go help packages'.

To run godoc with specific options, run godoc itself.

See also: go fix, go fmt, go vet.

Print Go environment information

Usage:

```
go env [var ...]
```

Env prints Go environment information.

By default env prints information as a shell script (on Windows, a batch file). If one or more variable names is given as arguments, env prints the value of each named variable on its own line.

Run go tool fix on packages

Usage:

```
go fix [packages]
```

Fix runs the Go fix command on the packages named by the import paths.

For more about fix, see 'godoc fix'. For more about specifying packages, see 'go help packages'.

To run fix with specific options, run 'go tool fix'.

See also: go fmt, go vet.

Run gofmt on package sources

Usage:

```
go fmt [packages]
```

Fmt runs the command 'gofmt -l -w' on the packages named by the import paths. It prints the names of the files that are modified.

For more about gofmt, see 'godoc gofmt'. For more about specifying packages, see 'go help packages'.

To run gofmt with specific options, run gofmt itself.

See also: go doc, go fix, go vet.

Download and install packages and dependencies

Usage:

```
go get [-a] [-d] [-fix] [-n] [-p n] [-u] [-v] [-x] [packages]
```

Get downloads and installs the packages named by the import paths, along with their dependencies.

The -a, -n, -v, -x, and -p flags have the same meaning as in 'go build' and 'go install'. See 'go help build'.

The -d flag instructs get to stop after downloading the packages; that is, it instructs get not to install the packages.

The -fix flag instructs get to run the fix tool on the downloaded packages before resolving dependencies or building the code.

The -u flag instructs get to use the network to update the named packages and their dependencies. By default, get uses the network to check out missing

packages but does not use it to look for updates to existing packages.

When checking out or updating a package, `get` looks for a branch or tag that matches the locally installed version of Go. The most important rule is that if the local installation is running version "go1", `get` searches for a branch or tag named "go1". If no such version exists it retrieves the most recent version of the package.

For more about specifying packages, see '`go help packages`'.

For more about how '`go get`' finds source code to download, see '`go help remote`'.

See also: `go build`, `go install`, `go clean`.

Compile and install packages and dependencies

Usage:

```
go install [build flags] [packages]
```

`Install` compiles and installs the packages named by the import paths, along with their dependencies.

For more about the build flags, see '`go help build`'. For more about specifying packages, see '`go help packages`'.

See also: `go build`, `go get`, `go clean`.

List packages

Usage:

```
go list [-e] [-f format] [-json] [packages]
```

`List` lists the packages named by the import paths, one per line.

The default output shows the package import path:

```
code.google.com/p/google-api-go-client/books/v1
code.google.com/p/goauth2/oauth
code.google.com/p/sqlite
```

The `-f` flag specifies an alternate format for the list, using the syntax of package template. The default output is equivalent to `-f '{{.ImportPath}}'`. The struct being passed to the template is:

```
type Package struct {
    Dir          string // directory containing package sources
    ImportPath   string // import path of package in dir
    Name         string // package name
    Doc          string // package documentation string
    Target       string // install path
    Goroot       bool    // is this package in the Go root?
    Standard     bool    // is this package part of the standard Go lib
    Stale        bool    // would 'go install' do anything for this pac
    Root         string // Go root or Go path dir containing this pack

    // Source files
    GoFiles      []string // .go source files (excluding CgoFiles, Test
    CgoFiles     []string // .go sources files that import "C"
    CFiles       []string // .c source files
    HFiles       []string // .h source files
    SFiles       []string // .s source files
    SysoFiles    []string // .syso object files to add to archive

    // Cgo directives
    CgoCFLAGS    []string // cgo: flags for C compiler
    CgoLDFLAGS   []string // cgo: flags for linker
    CgoPkgConfig []string // cgo: pkg-config names

    // Dependency information
    Imports      []string // import paths used by this package
    Deps         []string // all (recursively) imported dependencies

    // Error information
    Incomplete bool           // this package or a dependency has a
    Error       *PackageError // error loading package
    DepsErrors  []*PackageError // errors loading dependencies

    TestGoFiles  []string // _test.go files in package
    TestImports  []string // imports from TestGoFiles
    XTestGoFiles []string // _test.go files outside package
    XTestImports []string // imports from XTestGoFiles
}
```

The `-json` flag causes the package data to be printed in JSON format instead of using the template format.

The `-e` flag changes the handling of erroneous packages, those that cannot be

found or are malformed. By default, the list command prints an error to standard error for each erroneous package and omits the packages from consideration during the usual printing. With the `-e` flag, the list command never prints errors to standard error and instead processes the erroneous packages with the usual printing. Erroneous packages will have a non-empty `ImportPath` and a non-nil `Error` field; other information may or may not be missing (zeroed).

For more about specifying packages, see 'go help packages'.

Compile and run Go program

Usage:

```
go run [build flags] gofiles... [arguments...]
```

Run compiles and runs the main package comprising the named Go source files.

For more about build flags, see 'go help build'.

See also: go build.

Test packages

Usage:

```
go test [-c] [-i] [build flags] [packages] [flags for test binary]
```

'Go test' automates testing the packages named by the import paths. It prints a summary of the test results in the format:

```
ok    archive/tar    0.011s
FAIL  archive/zip    0.022s
ok    compress/gzip  0.033s
...
```

followed by detailed output for each failed package.

'Go test' recompiles each package along with any files with names matching the file pattern `"*_test.go"`. These additional files can contain test functions, benchmark functions, and example functions. See 'go help testfunc' for more.

By default, `go test` needs no arguments. It compiles and tests the package with source in the current directory, including tests, and runs the tests.

The package is built in a temporary directory so it does not interfere with the non-test installation.

In addition to the build flags, the flags handled by `'go test'` itself are:

`-c` Compile the test binary to `pkg.test` but do not run it.

`-i` Install packages that are dependencies of the test.
Do not run the test.

The test binary also accepts flags that control execution of the test; these flags are also accessible by `'go test'`. See `'go help testflag'` for details.

For more about build flags, see `'go help build'`. For more about specifying packages, see `'go help packages'`.

See also: `go build`, `go vet`.

Run specified go tool

Usage:

```
go tool [-n] command [args...]
```

Tool runs the `go tool` command identified by the arguments. With no arguments it prints the list of known tools.

The `-n` flag causes `tool` to print the command that would be executed but not execute it.

For more about each tool command, see `'go tool command -h'`.

Print Go version

Usage:

```
go version
```

Version prints the Go version, as reported by runtime.Version.

Run go tool vet on packages

Usage:

```
go vet [packages]
```

Vet runs the Go vet command on the packages named by the import paths.

For more about vet, see 'godoc vet'. For more about specifying packages, see 'go help packages'.

To run the vet tool with specific options, run 'go tool vet'.

See also: go fmt, go fix.

GOPATH environment variable

The Go path is used to resolve import statements. It is implemented by and documented in the go/build package.

The GOPATH environment variable lists places to look for Go code. On Unix, the value is a colon-separated string. On Windows, the value is a semicolon-separated string. On Plan 9, the value is a list.

GOPATH must be set to build and install packages outside the standard Go tree.

Each directory listed in GOPATH must have a prescribed structure:

The src/ directory holds source code. The path below 'src' determines the import path or executable name.

The pkg/ directory holds installed package objects. As in the Go tree, each target operating system and architecture pair has its own subdirectory of pkg (pkg/GOOS_GOARCH).

If DIR is a directory listed in the GOPATH, a package with source in DIR/src/foo/bar can be imported as "foo/bar" and has its compiled form installed to "DIR/pkg/GOOS_GOARCH/foo/bar.a".

The bin/ directory holds compiled commands. Each command is named for its source directory, but only the final element, not the entire path. That is, the command with source in DIR/src/foo/quux is installed into DIR/bin/quux, not DIR/bin/foo/quux. The foo/ is stripped so that you can add DIR/bin to your PATH to get at the installed commands. If the GOBIN environment variable is set, commands are installed to the directory it names instead of DIR/bin.

Here's an example directory layout:

```
GOPATH=/home/user/gocode
```

```
/home/user/gocode/  
  src/  
    foo/  
      bar/                (go code in package bar)  
        x.go  
      quux/              (go code in package main)  
        y.go  
  bin/  
    quux                 (installed command)  
  pkg/  
    linux_amd64/  
      foo/  
        bar.a            (installed package object)
```

Go searches each directory listed in GOPATH to find source code, but new packages are always downloaded into the first directory in the list.

Description of package lists

Many commands apply to a set of packages:

```
go action [packages]
```

Usually, [packages] is a list of import paths.

An import path that is a rooted path or that begins with a . or .. element is interpreted as a file system path and denotes the package in that directory.

Otherwise, the import path P denotes the package found in the directory DIR/src/P for some DIR listed in the GOPATH environment variable (see 'go help gopath').

If no import paths are given, the action applies to the package in the current directory.

The special import path "all" expands to all package directories found in all the GOPATH trees. For example, 'go list all' lists all the packages on the local system.

The special import path "std" is like all but expands to just the packages in the standard Go library.

An import path is a pattern if it includes one or more "..." wildcards, each of which can match any string, including the empty string and strings containing slashes. Such a pattern expands to all package directories found in the GOPATH trees with names matching the patterns. As a special case, x/... matches x as well as x's subdirectories. For example, net/... expands to net and packages in its subdirectories.

An import path can also name a package to be downloaded from a remote repository. Run 'go help remote' for details.

Every package in a program must have a unique import path. By convention, this is arranged by starting each path with a unique prefix that belongs to you. For example, paths used internally at Google all begin with 'google', and paths denoting remote repositories begin with the path to the code, such as 'code.google.com/p/project'.

As a special case, if the package list is a list of .go files from a single directory, the command is applied to a single synthesized package made up of exactly those files, ignoring any build constraints in those files and ignoring any other files in the directory.

Remote import path syntax

An import path (see 'go help importpath') denotes a package stored in the local file system. Certain import paths also describe how to obtain the source code for the package using a revision control system.

A few common code hosting sites have special syntax:

BitBucket (Mercurial)

```
import "bitbucket.org/user/project"  
import "bitbucket.org/user/project/sub/directory"
```

GitHub (Git)

```
import "github.com/user/project"  
import "github.com/user/project/sub/directory"
```

Google Code Project Hosting (Git, Mercurial, Subversion)

```
import "code.google.com/p/project"  
import "code.google.com/p/project/sub/directory"  
  
import "code.google.com/p/project.subrepository"  
import "code.google.com/p/project.subrepository/sub/director"
```

Launchpad (Bazaar)

```
import "launchpad.net/project"  
import "launchpad.net/project/series"  
import "launchpad.net/project/series/sub/directory"  
  
import "launchpad.net/~user/project/branch"  
import "launchpad.net/~user/project/branch/sub/directory"
```

For code hosted on other servers, import paths may either be qualified with the version control type, or the go tool can dynamically fetch the import path over https/http and discover where the code resides from a <meta> tag in the HTML.

To declare the code location, an import path of the form

```
repository.vcs/path
```

specifies the given repository, with or without the .vcs suffix, using the named version control system, and then the path inside that repository. The supported version control systems are:

Bazaar	.bzd
Git	.git
Mercurial	.hg
Subversion	.svn

For example,

```
import "example.org/user/foo.hg"
```

denotes the root directory of the Mercurial repository at example.org/user/foo or foo.hg, and

```
import "example.org/repo.git/foo/bar"
```

denotes the foo/bar directory of the Git repository at example.com/repo or repo.git.

When a version control system supports multiple protocols, each is tried in turn when downloading. For example, a Git download tries git://, then https://, then http://.

If the import path is not a known code hosting site and also lacks a version control qualifier, the go tool attempts to fetch the import over https/http and looks for a <meta> tag in the document's HTML <head>.

The meta tag has the form:

```
<meta name="go-import" content="import-prefix vcs repo-root">
```

The import-prefix is the import path corresponding to the repository root. It must be a prefix or an exact match of the package being fetched with "go get". If it's not an exact match, another http request is made at the prefix to verify the <meta> tags match.

The vcs is one of "git", "hg", "svn", etc,

The repo-root is the root of the version control system containing a scheme and not containing a .vcs qualifier.

For example,

```
import "example.org/pkg/foo"
```

will result in the following request(s):

<https://example.org/pkg/foo?go-get=1> (preferred)
<http://example.org/pkg/foo?go-get=1> (fallback)

If that page contains the meta tag

```
<meta name="go-import" content="example.org git https://code.org/r/p
```

the go tool will verify that <https://example.org/?go-get=1> contains the same meta tag and then git clone <https://code.org/r/p/exproj> into GOPATH/src/example.org.

New downloaded packages are written to the first directory listed in the GOPATH environment variable (see 'go help gopath').

The go command attempts to download the version of the package appropriate for the Go release being used. Run 'go help install' for more.

Description of testing flags

The 'go test' command takes both flags that apply to 'go test' itself and flags that apply to the resulting test binary.

The test binary, called pkg.test, where pkg is the name of the directory containing the package sources, has its own flags:

- test.v
Verbose output: log all tests as they are run.
- test.run pattern
Run only those tests and examples matching the regular expression.
- test.bench pattern
Run benchmarks matching the regular expression.
By default, no benchmarks run.
- test.cpuprofile cpu.out
Write a CPU profile to the specified file before exiting.
- test.memprofile mem.out
Write a memory profile to the specified file when all tests are complete.
- test.memprofilerate n
Enable more precise (and expensive) memory profiles by setting runtime.MemProfileRate. See 'godoc runtime MemProfileRate'.
To profile all memory allocations, use -test.memprofilerate=1 and set the environment variable GOGC=off to disable the garbage collector, provided the test can run in the available memory without garbage collection.
- test.parallel n
Allow parallel execution of test functions that call t.Parallel.

The value of this flag is the maximum number of tests to run simultaneously; by default, it is set to the value of GOMAXPROCS

`-test.short`

Tell long-running tests to shorten their run time. It is off by default but set during `all.bash` so that installing the Go tree can run a sanity check but not spend time running exhaustive tests.

`-test.timeout t`

If a test runs longer than `t`, panic.

`-test.benchmark n`

Run enough iterations of each benchmark to take `n` seconds. The default is 1 second.

`-test.cpu 1,2,4`

Specify a list of GOMAXPROCS values for which the tests or benchmarks should be executed. The default is the current value of GOMAXPROCS.

For convenience, each of these `-test.X` flags of the test binary is also available as the flag `-X` in 'go test' itself. Flags not listed here are passed through unaltered. For instance, the command

```
go test -x -v -cpuprofile=prof.out -dir=testdata -update
```

will compile the test binary and then run it as

```
pkg.test -test.v -test.cpuprofile=prof.out -dir=testdata -update
```

Description of testing functions

The 'go test' command expects to find test, benchmark, and example functions in the `"*_test.go"` files corresponding to the package under test.

A test function is one named `TestXXX` (where `XXX` is any alphanumeric string not starting with a lower case letter) and should have the signature,

```
func TestXXX(t *testing.T) { ... }
```

A benchmark function is one named `BenchmarkXXX` and should have the signature,

```
func BenchmarkXXX(b *testing.B) { ... }
```

An example function is similar to a test function but, instead of using `*testing.T` to report success or failure, prints output to `os.Stdout` and `os.Stderr`. That output is compared against the function's "Output:" comment, which must be the last comment in the function body (see example below). An example with no such comment, or with no text after "Output:" is compiled but not executed.

Godoc displays the body of `ExampleXXX` to demonstrate the use of the function, constant, or variable `XXX`. An example of a method `M` with receiver type `T` or `*T` is named `ExampleT_M`. There may be multiple examples for a given function, constant, or variable, distinguished by a trailing `_xxx`, where `xxx` is a suffix not beginning with an upper case letter.

Here is an example of an example:

```
func ExamplePrintln() {
    Println("The output of\nthis example.")
    // Output: The output of
    // this example.
}
```

The entire test file is presented as the example when it contains a single example function, at least one other function, type, variable, or constant declaration, and no test or benchmark functions.

See the documentation of the testing package for more information.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

FAQ

Origins

What is the purpose of the project?

No major systems language has emerged in over a decade, but over that time the computing landscape has changed tremendously. There are several trends:

- Computers are enormously quicker but software development is not faster.
- Dependency management is a big part of software development today but the “header files” of languages in the C tradition are antithetical to clean dependency analysis—and fast compilation.
- There is a growing rebellion against cumbersome type systems like those of Java and C++, pushing people towards dynamically typed languages such as Python and JavaScript.
- Some fundamental concepts such as garbage collection and parallel computation are not well supported by popular systems languages.
- The emergence of multicore computers has generated worry and confusion.

We believe it's worth trying again with a new language, a concurrent, garbage-collected language with fast compilation. Regarding the points above:

- It is possible to compile a large Go program in a few seconds on a single computer.
- Go provides a model for software construction that makes dependency analysis easy and avoids much of the overhead of C-style include files and libraries.
- Go's type system has no hierarchy, so no time is spent defining the relationships between types. Also, although Go has static types the language attempts to make types feel lighter weight than in typical OO languages.
- Go is fully garbage-collected and provides fundamental support for concurrent execution and communication.
- By its design, Go proposes an approach for the construction of system software on multicore machines.

What is the origin of the name?

“Ogle” would be a good name for a Go debugger.

What's the origin of the mascot?

The mascot and logo were designed by [Renée French](#), who also designed [Glenda](#), the Plan 9 bunny. The gopher is derived from one she used for an [WFMU](#) T-shirt design some years ago. The logo and mascot are covered by the [Creative Commons Attribution 3.0](#) license.

What is the history of the project?

Robert Griesemer, Rob Pike and Ken Thompson started sketching the goals for a new language on the white board on September 21, 2007. Within a few days the goals had settled into a plan to do something and a fair idea of what it would be. Design continued part-time in parallel with unrelated work. By January 2008, Ken had started work on a compiler with which to explore ideas; it generated C code as its output. By mid-year the language had become a full-time project and had settled enough to attempt a production compiler. In May 2008, Ian Taylor independently started on a GCC front end for Go using the draft specification. Russ Cox joined in late 2008 and helped move the language and libraries from prototype to reality.

Go became a public open source project on November 10, 2009. Many people from the community have contributed ideas, discussions, and code.

Why are you creating a new language?

Go was born out of frustration with existing languages and environments for systems programming. Programming had become too difficult and the choice of languages was partly to blame. One had to choose either efficient compilation, efficient execution, or ease of programming; all three were not available in the same mainstream language. Programmers who could were choosing ease over safety and efficiency by moving to dynamically typed languages such as Python and JavaScript rather than C++ or, to a lesser extent, Java.

Go is an attempt to combine the ease of programming of an interpreted, dynamically typed language with the efficiency and safety of a statically typed, compiled language. It also aims to be modern, with support for networked and multicore computing. Finally, it is intended to be *fast*: it should take at most a

few seconds to build a large executable on a single computer. To meet these goals required addressing a number of linguistic issues: an expressive but lightweight type system; concurrency and garbage collection; rigid dependency specification; and so on. These cannot be addressed well by libraries or tools; a new language was called for.

What are Go's ancestors?

Go is mostly in the C family (basic syntax), with significant input from the Pascal/Modula/Oberon family (declarations, packages), plus some ideas from languages inspired by Tony Hoare's CSP, such as Newsqueak and Limbo (concurrency). However, it is a new language across the board. In every respect the language was designed by thinking about what programmers do and how to make programming, at least the kind of programming we do, more effective, which means more fun.

What are the guiding principles in the design?

Programming today involves too much bookkeeping, repetition, and clerical work. As Dick Gabriel says, “Old programs read like quiet conversations between a well-spoken research worker and a well-studied mechanical colleague, not as a debate with a compiler. Who'd have guessed sophistication bought such noise?” The sophistication is worthwhile—no one wants to go back to the old languages—but can it be more quietly achieved?

Go attempts to reduce the amount of typing in both senses of the word. Throughout its design, we have tried to reduce clutter and complexity. There are no forward declarations and no header files; everything is declared exactly once. Initialization is expressive, automatic, and easy to use. Syntax is clean and light on keywords. Stuttering (`foo.Foo* myFoo = new(foo.Foo)`) is reduced by simple type derivation using the `:=` declare-and-initialize construct. And perhaps most radically, there is no type hierarchy: types just *are*, they don't have to announce their relationships. These simplifications allow Go to be expressive yet comprehensible without sacrificing, well, sophistication.

Another important principle is to keep the concepts orthogonal. Methods can be implemented for any type; structures represent data while interfaces represent abstraction; and so on. Orthogonality makes it easier to understand what happens when things combine.

Usage

Is Google using Go internally?

Yes. There are now several Go programs deployed in production inside Google. A public example is the server behind <http://golang.org>. It's just the [godoc](#) document server running in a production configuration on [Google App Engine](#).

Do Go programs link with C/C++ programs?

There are two Go compiler implementations, gc (the 6g program and friends) and gccgo. Gc uses a different calling convention and linker and can therefore only be linked with C programs using the same convention. There is such a C compiler but no C++ compiler. gccgo is a GCC front-end that can, with care, be linked with GCC-compiled C or C++ programs.

The [cgo](#) program provides the mechanism for a “foreign function interface” to allow safe calling of C libraries from Go code. SWIG extends this capability to C++ libraries.

Does Go support Google's protocol buffers?

A separate open source project provides the necessary compiler plugin and library. It is available at <http://code.google.com/p/goprotobuf/>

Can I translate the Go home page into another language?

Absolutely. We encourage developers to make Go Language sites in their own languages. However, if you choose to add the Google logo or branding to your site (it does not appear on golang.org), you will need to abide by the guidelines at <http://www.google.com/permissions/guidelines.html>

Design

What's up with Unicode identifiers?

It was important to us to extend the space of identifiers from the confines of ASCII. Go's rule—identifier characters must be letters or digits as defined by Unicode—is simple to understand and to implement but has restrictions. Combining characters are excluded by design, for instance. Until there is an agreed external definition of what an identifier might be, plus a definition of canonicalization of identifiers that guarantees no ambiguity, it seemed better to keep combining characters out of the mix. Thus we have a simple rule that can be expanded later without breaking programs, one that avoids bugs that would surely arise from a rule that admits ambiguous identifiers.

On a related note, since an exported identifier must begin with an upper-case letter, identifiers created from “letters” in some languages can, by definition, not be exported. For now the only solution is to use something like `x0000Z`, which is clearly unsatisfactory; we are considering other options. The case-for-visibility rule is unlikely to change however; it's one of our favorite features of Go.

Why does Go not have feature X?

Every language contains novel features and omits someone's favorite feature. Go was designed with an eye on felicity of programming, speed of compilation, orthogonality of concepts, and the need to support features such as concurrency and garbage collection. Your favorite feature may be missing because it doesn't fit, because it affects compilation speed or clarity of design, or because it would make the fundamental system model too difficult.

If it bothers you that Go is missing feature *X*, please forgive us and investigate the features that Go does have. You might find that they compensate in interesting ways for the lack of *X*.

Why does Go not have generic types?

Generics may well be added at some point. We don't feel an urgency for them, although we understand some programmers do.

Generics are convenient but they come at a cost in complexity in the type system and run-time. We haven't yet found a design that gives value proportionate to the complexity, although we continue to think about it. Meanwhile, Go's built-in maps and slices, plus the ability to use the empty interface to construct containers (with explicit unboxing) mean in many cases it is possible to write code that does what generics would enable, if less smoothly.

This remains an open issue.

Why does Go not have exceptions?

We believe that coupling exceptions to a control structure, as in the try-catch-finally idiom, results in convoluted code. It also tends to encourage programmers to label too many ordinary errors, such as failing to open a file, as exceptional.

Go takes a different approach. For plain error handling, Go's multi-value returns make it easy to report an error without overloading the return value. [A canonical error type, coupled with Go's other features](#), makes error handling pleasant but quite different from that in other languages.

Go also has a couple of built-in functions to signal and recover from truly exceptional conditions. The recovery mechanism is executed only as part of a function's state being torn down after an error, which is sufficient to handle catastrophe but requires no extra control structures and, when used well, can result in clean error-handling code.

See the [Defer, Panic, and Recover](#) article for details.

Why does Go not have assertions?

Go doesn't provide assertions. They are undeniably convenient, but our experience has been that programmers use them as a crutch to avoid thinking about proper error handling and reporting. Proper error handling means that servers continue operation after non-fatal errors instead of crashing. Proper error reporting means that errors are direct and to the point, saving the programmer from interpreting a large crash trace. Precise errors are particularly important when the programmer seeing the errors is not familiar with the code.

We understand that this is a point of contention. There are many things in the Go language and libraries that differ from modern practices, simply because we feel it's sometimes worth trying a different approach.

Why build concurrency on the ideas of CSP?

Concurrency and multi-threaded programming have a reputation for difficulty. We believe this is due partly to complex designs such as pthreads and partly to overemphasis on low-level details such as mutexes, condition variables, and memory barriers. Higher-level interfaces enable much simpler code, even if there are still mutexes and such under the covers.

One of the most successful models for providing high-level linguistic support for concurrency comes from Hoare's Communicating Sequential Processes, or CSP. Occam and Erlang are two well known languages that stem from CSP. Go's concurrency primitives derive from a different part of the family tree whose main contribution is the powerful notion of channels as first class objects.

Why goroutines instead of threads?

Goroutines are part of making concurrency easy to use. The idea, which has been around for a while, is to multiplex independently executing functions—coroutines—onto a set of threads. When a coroutine blocks, such as by calling a blocking system call, the run-time automatically moves other coroutines on the same operating system thread to a different, runnable thread so they won't be blocked. The programmer sees none of this, which is the point. The result, which we call goroutines, can be very cheap: unless they spend a lot of time in long-running system calls, they cost little more than the memory for the stack, which is just a few kilobytes.

To make the stacks small, Go's run-time uses segmented stacks. A newly minted goroutine is given a few kilobytes, which is almost always enough. When it isn't, the run-time allocates (and frees) extension segments automatically. The overhead averages about three cheap instructions per function call. It is practical to create hundreds of thousands of goroutines in the same address space. If goroutines were just threads, system resources would run out at a much smaller number.

Why are map operations not defined to be atomic?

After long discussion it was decided that the typical use of maps did not require safe access from multiple threads, and in those cases where it did, the map was probably part of some larger data structure or computation that was already synchronized. Therefore requiring that all map operations grab a mutex would slow down most programs and add safety to few. This was not an easy decision, however, since it means uncontrolled map access can crash the program.

The language does not preclude atomic map updates. When required, such as when hosting an untrusted program, the implementation could interlock map access.

Types

Is Go an object-oriented language?

Yes and no. Although Go has types and methods and allows an object-oriented style of programming, there is no type hierarchy. The concept of “interface” in Go provides a different approach that we believe is easy to use and in some ways more general. There are also ways to embed types in other types to provide something analogous—but not identical—to subclassing. Moreover, methods in Go are more general than in C++ or Java: they can be defined for any sort of data, even built-in types such as plain, “unboxed” integers. They are not restricted to structs (classes).

Also, the lack of type hierarchy makes “objects” in Go feel much more lightweight than in languages such as C++ or Java.

How do I get dynamic dispatch of methods?

The only way to have dynamically dispatched methods is through an interface. Methods on a struct or any other concrete type are always resolved statically.

Why is there no type inheritance?

Object-oriented programming, at least in the best-known languages, involves too much discussion of the relationships between types, relationships that often could be derived automatically. Go takes a different approach.

Rather than requiring the programmer to declare ahead of time that two types are related, in Go a type automatically satisfies any interface that specifies a subset of its methods. Besides reducing the bookkeeping, this approach has real advantages. Types can satisfy many interfaces at once, without the complexities of traditional multiple inheritance. Interfaces can be very lightweight—an interface with one or even zero methods can express a useful concept. Interfaces can be added after the fact if a new idea comes along or for testing—without annotating the original types. Because there are no explicit relationships between types and interfaces, there is no type hierarchy to manage or discuss.

It's possible to use these ideas to construct something analogous to type-safe Unix pipes. For instance, see how `fmt.Fprintf` enables formatted printing to any output, not just a file, or how the `bufio` package can be completely separate from file I/O, or how the `image` packages generate compressed image files. All these ideas stem from a single interface (`io.Writer`) representing a single method (`write`). And that's only scratching the surface. Go's interfaces have a profound influence on how programs are structured.

It takes some getting used to but this implicit style of type dependency is one of the most productive things about Go.

Why is `len` a function and not a method?

We debated this issue but decided implementing `len` and friends as functions was fine in practice and didn't complicate questions about the interface (in the Go type sense) of basic types.

Why does Go not support overloading of methods and operators?

Method dispatch is simplified if it doesn't need to do type matching as well. Experience with other languages told us that having a variety of methods with the same name but different signatures was occasionally useful but that it could also be confusing and fragile in practice. Matching only by name and requiring consistency in the types was a major simplifying decision in Go's type system.

Regarding operator overloading, it seems more a convenience than an absolute requirement. Again, things are simpler without it.

Why doesn't Go have "implements" declarations?

A Go type satisfies an interface by implementing the methods of that interface, nothing more. This property allows interfaces to be defined and used without having to modify existing code. It enables a kind of "duck typing" that promotes separation of concerns and improves code re-use, and makes it easier to build on patterns that emerge as the code develops. The semantics of interfaces is one of the main reasons for Go's nimble, lightweight feel.

See the [question on type inheritance](#) for more detail.

How can I guarantee my type satisfies an interface?

You can ask the compiler to check that the type `T` implements the interface `I` by attempting an assignment:

```
type T struct{}
var _ I = T{} // Verify that T implements I.
```

If `T` doesn't implement `I`, the mistake will be caught at compile time.

If you wish the users of an interface to explicitly declare that they implement it, you can add a method with a descriptive name to the interface's method set. For example:

```
type Fooer interface {
    Foo()
    ImplementsFooer()
}
```

A type must then implement the `ImplementsFooer` method to be a `Fooer`, clearly documenting the fact and announcing it in [godoc](#)'s output.

```
type Bar struct{}
func (b Bar) ImplementsFooer() {}
func (b Bar) Foo() {}
```

Most code doesn't make use of such constraints, since they limit the utility of the interface idea. Sometimes, though, they're necessary to resolve ambiguities among similar interfaces.

Why doesn't type `T` satisfy the `Equal` interface?

Consider this simple interface to represent an object that can compare itself with another value:

```
type Equaler interface {
    Equal(Equaler) bool
}
```

and this type, `T`:

```
type T int
func (t T) Equal(u T) bool { return t == u } // does not satisfy Equ
```

Unlike the analogous situation in some polymorphic type systems, τ does not implement `Equaler`. The argument type of `τ .Equal` is τ , not literally the required type `Equaler`.

In Go, the type system does not promote the argument of `Equal`; that is the programmer's responsibility, as illustrated by the type `T2`, which does implement `Equaler`:

```
type T2 int
func (t T2) Equal(u Equaler) bool { return t == u.(T2) } // satisfi
```

Even this isn't like other type systems, though, because in Go *any* type that satisfies `Equaler` could be passed as the argument to `T2.Equal`, and at run time we must check that the argument is of type `T2`. Some languages arrange to make that guarantee at compile time.

A related example goes the other way:

```
type Opener interface {
    Open(name) Reader
}

func (t T3) Open() *os.File
```

In Go, `T3` does not satisfy `Opener`, although it might in another language.

While it is true that Go's type system does less for the programmer in such cases, the lack of subtyping makes the rules about interface satisfaction very easy to state: are the function's names and signatures exactly those of the interface? Go's rule is also easy to implement efficiently. We feel these benefits offset the lack of automatic type promotion. Should Go one day adopt some form of generic typing, we expect there would be a way to express the idea of these examples and also have them be statically checked.

Can I convert a `[]T` to an `[]interface{}`?

Not directly, because they do not have the same representation in memory. It is necessary to copy the elements individually to the destination slice. This example converts a slice of `int` to a slice of `interface{}`:

```
t := []int{1, 2, 3, 4}
s := make([]interface{}, len(t))
```

```
for i, v := range t {
    s[i] = v
}
```

Why is my nil error value not equal to nil?

Under the covers, interfaces are implemented as two elements, a type and a value. The value, called the interface's dynamic value, is an arbitrary concrete value and the type is that of the value. For the `int` value 3, an interface value contains, schematically, `(int, 3)`.

An interface value is `nil` only if the inner value and type are both unset, `(nil, nil)`. In particular, a `nil` interface will always hold a `nil` type. If we store a pointer of type `*int` inside an interface value, the inner type will be `*int` regardless of the value of the pointer: `(*int, nil)`. Such an interface value will therefore be non-`nil` *even when the pointer inside is nil*.

This situation can be confusing, and often arises when a `nil` value is stored inside an interface value such as an error return:

```
func returnsError() error {
    var p *MyError = nil
    if bad() {
        p = ErrBad
    }
    return p // Will always return a non-nil error.
}
```

If all goes well, the function returns a `nil` `p`, so the return value is an error interface value holding `(*MyError, nil)`. This means that if the caller compares the returned error to `nil`, it will always look as if there was an error even if nothing bad happened. To return a proper `nil` error to the caller, the function must return an explicit `nil`:

```
func returnsError() error {
    if bad() {
        return ErrBad
    }
    return nil
}
```

It's a good idea for functions that return errors always to use the error type in their signature (as we did above) rather than a concrete type such as `*MyError`, to

help guarantee the error is created correctly. As an example, [os.Open](#) returns an error even though, if not nil, it's always of concrete type [*os.PathError](#).

Similar situations to those described here can arise whenever interfaces are used. Just keep in mind that if any concrete value has been stored in the interface, the interface will not be nil. For more information, see [The Laws of Reflection](#).

Why are there no untagged unions, as in C?

Untagged unions would violate Go's memory safety guarantees.

Why does Go not have variant types?

Variant types, also known as algebraic types, provide a way to specify that a value might take one of a set of other types, but only those types. A common example in systems programming would specify that an error is, say, a network error, a security error or an application error and allow the caller to discriminate the source of the problem by examining the type of the error. Another example is a syntax tree in which each node can be a different type: declaration, statement, assignment and so on.

We considered adding variant types to Go, but after discussion decided to leave them out because they overlap in confusing ways with interfaces. What would happen if the elements of a variant type were themselves interfaces?

Also, some of what variant types address is already covered by the language. The error example is easy to express using an interface value to hold the error and a type switch to discriminate cases. The syntax tree example is also doable, although not as elegantly.

Values

Why does Go not provide implicit numeric conversions?

The convenience of automatic conversion between numeric types in C is outweighed by the confusion it causes. When is an expression unsigned? How big is the value? Does it overflow? Is the result portable, independent of the machine on which it executes? It also complicates the compiler; “the usual arithmetic conversions” are not easy to implement and inconsistent across architectures. For reasons of portability, we decided to make things clear and straightforward at the cost of some explicit conversions in the code. The definition of constants in Go—arbitrary precision values free of signedness and size annotations—ameliorates matters considerably, though.

A related detail is that, unlike in C, `int` and `int64` are distinct types even if `int` is a 64-bit type. The `int` type is generic; if you care about how many bits an integer holds, Go encourages you to be explicit.

Why are maps built in?

The same reason strings are: they are such a powerful and important data structure that providing one excellent implementation with syntactic support makes programming more pleasant. We believe that Go's implementation of maps is strong enough that it will serve for the vast majority of uses. If a specific application can benefit from a custom implementation, it's possible to write one but it will not be as convenient syntactically; this seems a reasonable tradeoff.

Why don't maps allow slices as keys?

Map lookup requires an equality operator, which slices do not implement. They don't implement equality because equality is not well defined on such types; there are multiple considerations involving shallow vs. deep comparison, pointer vs. value comparison, how to deal with recursive types, and so on. We may revisit this issue—and implementing equality for slices will not invalidate any existing programs—but without a clear idea of what equality of slices should mean, it was simpler to leave it out for now.

In Go 1, unlike prior releases, equality is defined for structs and arrays, so such types can be used as map keys. Slices still do not have a definition of equality, though.

Why are maps, slices, and channels references while arrays are values?

There's a lot of history on that topic. Early on, maps and channels were syntactically pointers and it was impossible to declare or use a non-pointer instance. Also, we struggled with how arrays should work. Eventually we decided that the strict separation of pointers and values made the language harder to use. Introducing reference types, including slices to handle the reference form of arrays, resolved these issues. Reference types add some regrettable complexity to the language but they have a large effect on usability: Go became a more productive, comfortable language when they were introduced.

Writing Code

How are libraries documented?

There is a program, `godoc`, written in Go, that extracts package documentation from the source code. It can be used on the command line or on the web. An instance is running at <http://golang.org/pkg/>. In fact, `godoc` implements the full site at <http://golang.org/>.

Is there a Go programming style guide?

Eventually, there may be a small number of rules to guide things like naming, layout, and file organization. The document [Effective Go](#) contains some style advice. More directly, the program `gofmt` is a pretty-printer whose purpose is to enforce layout rules; it replaces the usual compendium of do's and don'ts that allows interpretation. All the Go code in the repository has been run through `gofmt`.

How do I submit patches to the Go libraries?

The library sources are in `go/src/pkg`. If you want to make a significant change, please discuss on the mailing list before embarking.

See the document [Contributing to the Go project](#) for more information about how to proceed.

Pointers and Allocation

When are function parameters passed by value?

As in all languages in the C family, everything in Go is passed by value. That is, a function always gets a copy of the thing being passed, as if there were an assignment statement assigning the value to the parameter. For instance, passing an `int` value to a function makes a copy of the `int`, and passing a pointer value makes a copy of the pointer, but not the data it points to. (See the next section for a discussion of how this affects method receivers.)

Map and slice values behave like pointers: they are descriptors that contain pointers to the underlying map or slice data. Copying a map or slice value doesn't copy the data it points to. Copying an interface value makes a copy of the thing stored in the interface value. If the interface value holds a struct, copying the interface value makes a copy of the struct. If the interface value holds a pointer, copying the interface value makes a copy of the pointer, but again not the data it points to.

Should I define methods on values or pointers?

```
func (s *MyStruct) pointerMethod() { } // method on pointer
func (s MyStruct) valueMethod()   { } // method on value
```

For programmers unaccustomed to pointers, the distinction between these two examples can be confusing, but the situation is actually very simple. When defining a method on a type, the receiver (`s` in the above examples) behaves exactly as if it were an argument to the method. Whether to define the receiver as a value or as a pointer is the same question, then, as whether a function argument should be a value or a pointer. There are several considerations.

First, and most important, does the method need to modify the receiver? If it does, the receiver *must* be a pointer. (Slices and maps are reference types, so their story is a little more subtle, but for instance to change the length of a slice in a method the receiver must still be a pointer.) In the examples above, if `pointerMethod` modifies the fields of `s`, the caller will see those changes, but `valueMethod` is called with a copy of the caller's argument (that's the definition of passing a value), so changes it makes will be invisible to the caller.

By the way, pointer receivers are identical to the situation in Java, although in Java the pointers are hidden under the covers; it's Go's value receivers that are unusual.

Second is the consideration of efficiency. If the receiver is large, a big struct for instance, it will be much cheaper to use a pointer receiver.

Next is consistency. If some of the methods of the type must have pointer receivers, the rest should too, so the method set is consistent regardless of how the type is used. See the section on [method sets](#) for details.

For types such as basic types, slices, and small structs, a value receiver is very cheap so unless the semantics of the method requires a pointer, a value receiver is efficient and clear.

What's the difference between new and make?

In short: `new` allocates memory, `make` initializes the slice, map, and channel types.

See the [relevant section of Effective Go](#) for more details.

Why is int 32 bits on 64 bit machines?

The sizes of `int` and `uint` are implementation-specific but the same as each other on a given platform. The 64 bit Go compilers (both `gc` and `gccgo`) use a 32 bit representation for `int`. Code that relies on a particular size of value should use an explicitly sized type, like `int64`. On the other hand, floating-point scalars and complex numbers are always sized: `float32`, `complex64`, etc., because programmers should be aware of precision when using floating-point numbers. The default size of a floating-point constant is `float64`.

At the moment, all implementations use 32-bit ints, an essentially arbitrary decision. However, we expect that `int` will be increased to 64 bits on 64-bit architectures in a future release of Go.

How do I know whether a variable is allocated on the heap or the stack?

From a correctness standpoint, you don't need to know. Each variable in Go exists as long as there are references to it. The storage location chosen by the implementation is irrelevant to the semantics of the language.

The storage location does have an effect on writing efficient programs. When possible, the Go compilers will allocate variables that are local to a function in that function's stack frame. However, if the compiler cannot prove that the variable is not referenced after the function returns, then the compiler must allocate the variable on the garbage-collected heap to avoid dangling pointer errors. Also, if a local variable is very large, it might make more sense to store it on the heap rather than the stack.

In the current compilers, if a variable has its address taken, that variable is a candidate for allocation on the heap. However, a basic *escape analysis* recognizes some cases when such variables will not live past the return from the function and can reside on the stack.

Concurrency

What operations are atomic? What about mutexes?

We haven't fully defined it all yet, but some details about atomicity are available in the [Go Memory Model specification](#).

Regarding mutexes, the [sync](#) package implements them, but we hope Go programming style will encourage people to try higher-level techniques. In particular, consider structuring your program so that only one goroutine at a time is ever responsible for a particular piece of data.

Do not communicate by sharing memory. Instead, share memory by communicating.

See the [Share Memory By Communicating](#) code walk and its [associated article](#) for a detailed discussion of this concept.

Why doesn't my multi-goroutine program use multiple CPUs?

You must set the `GOMAXPROCS` shell environment variable or use the similarly-named [function](#) of the runtime package to allow the run-time support to utilize more than one OS thread.

Programs that perform parallel computation should benefit from an increase in `GOMAXPROCS`.

Why does using `GOMAXPROCS > 1` sometimes make my program slower?

It depends on the nature of your program. Problems that are intrinsically sequential cannot be sped up by adding more goroutines. Concurrency only becomes parallelism when the problem is intrinsically parallel.

In practical terms, programs that spend more time communicating on channels than doing computation will experience performance degradation when using multiple OS threads. This is because sending data between threads involves

switching contexts, which has significant cost. For instance, the [prime sieve example](#) from the Go specification has no significant parallelism although it launches many goroutines; increasing GOMAXPROCS is more likely to slow it down than to speed it up.

Go's goroutine scheduler is not as good as it needs to be. In future, it should recognize such cases and optimize its use of OS threads. For now, GOMAXPROCS should be set on a per-application basis.

Functions and Methods

Why do T and *T have different method sets?

From the [Go Spec](#):

The method set of any other named type τ consists of all methods with receiver type τ . The method set of the corresponding pointer type $*\tau$ is the set of all methods with receiver $*\tau$ or τ (that is, it also contains the method set of τ).

If an interface value contains a pointer $*\tau$, a method call can obtain a value by dereferencing the pointer, but if an interface value contains a value τ , there is no useful way for a method call to obtain a pointer.

Even in cases where the compiler could take the address of a value to pass to the method, if the method modifies the value the changes will be lost in the caller. As a common example, this code:

```
var buf bytes.Buffer
io.Copy(buf, os.Stdin)
```

would copy standard input into a *copy* of `buf`, not into `buf` itself. This is almost never the desired behavior.

What happens with closures running as goroutines?

Some confusion may arise when using closures with concurrency. Consider the following program:

```
func main() {
    done := make(chan bool)

    values := []string{"a", "b", "c"}
    for _, v := range values {
        go func() {
            fmt.Println(v)
            done <- true
        }()
    }
}
```

```
    // wait for all goroutines to complete before exiting
    for _ = range values {
        <-done
    }
}
```

One might mistakenly expect to see `a, b, c` as the output. What you'll probably see instead is `c, c, c`. This is because each iteration of the loop uses the same instance of the variable `v`, so each closure shares that single variable. When the closure runs, it prints the value of `v` at the time `fmt.Println` is executed, but `v` may have been modified since the goroutine was launched.

To bind the value of `v` to each closure as they are launched, one could modify the inner loop to read:

```
    for _, v := range values {
        go func(u string) {
            fmt.Println(u)
            done <- true
        }(v)
    }
```

In this example, the value of `v` is passed as an argument to the anonymous function. That value is then accessible inside the function as the variable `u`.

Control flow

Does Go have the `?:` operator?

There is no ternary form in Go. You may use the following to achieve the same result:

```
if expr {  
    n = trueVal  
} else {  
    n = falseVal  
}
```

Packages and Testing

How do I create a multifile package?

Put all the source files for the package in a directory by themselves. Source files can refer to items from different files at will; there is no need for forward declarations or a header file.

Other than being split into multiple files, the package will compile and test just like a single-file package.

How do I write a unit test?

Create a new file ending in `_test.go` in the same directory as your package sources. Inside that file, import `"testing"` and write functions of the form

```
func TestFoo(t *testing.T) {  
    ...  
}
```

Run `go test` in that directory. That script finds the `Test` functions, builds a test binary, and runs it.

See the [How to Write Go Code](#) document, the [testing](#) package and the [go test](#) subcommand for more details.

Where is my favorite helper function for testing?

Go's standard [testing](#) package makes it easy to write unit tests, but it lacks features provided in other language's testing frameworks such as assertion functions. An [earlier section](#) of this document explained why Go doesn't have assertions, and the same arguments apply to the use of `assert` in tests. Proper error handling means letting other tests run after one has failed, so that the person debugging the failure gets a complete picture of what is wrong. It is more useful for a test to report that `isPrime` gives the wrong answer for 2, 3, 5, and 7 (or for 2, 4, 8, and 16) than to report that `isPrime` gives the wrong answer for 2 and therefore no more tests were run. The programmer who triggers the test failure may not be familiar with the code that fails. Time invested writing a good

error message now pays off later when the test breaks.

A related point is that testing frameworks tend to develop into mini-languages of their own, with conditionals and controls and printing mechanisms, but Go already has all those capabilities; why recreate them? We'd rather write tests in Go; it's one fewer language to learn and the approach keeps the tests straightforward and easy to understand.

If the amount of extra code required to write good errors seems repetitive and overwhelming, the test might work better if table-driven, iterating over a list of inputs and outputs defined in a data structure (Go has excellent support for data structure literals). The work to write a good test and good error messages will then be amortized over many test cases. The standard Go library is full of illustrative examples, such as in [the formatting tests for the fmt package](#).

Implementation

What compiler technology is used to build the compilers?

`gccgo` has a C++ front-end with a recursive descent parser coupled to the standard GCC back end. `gc` is written in C using `yacc/bison` for the parser. Although it's a new program, it fits in the Plan 9 C compiler suite (<http://plan9.bell-labs.com/sys/doc/compiler.html>) and uses a variant of the Plan 9 loader to generate ELF/Mach-O/PE binaries.

We considered writing `gc`, the original Go compiler, in Go itself but elected not to do so because of the difficulties of bootstrapping and especially of open source distribution—you'd need a Go compiler to set up a Go environment. `gccgo`, which came later, makes it possible to consider writing a compiler in Go, which might well happen. (Go would be a fine language in which to implement a compiler; a native lexer and parser are already available in the `go` package.)

We also considered using LLVM for `gc` but we felt it was too large and slow to meet our performance goals.

How is the run-time support implemented?

Again due to bootstrapping issues, the run-time code is mostly in C (with a tiny bit of assembler) although Go is capable of implementing most of it now. `gccgo`'s run-time support uses `glibc`. `gc` uses a custom library to keep the footprint under control; it is compiled with a version of the Plan 9 C compiler that supports segmented stacks for goroutines. The `gccgo` compiler implements segmented stacks on Linux only, supported by recent modifications to the gold linker.

Why is my trivial program such a large binary?

The linkers in the `gc` tool chain (51, 61, and 81) do static linking. All Go binaries therefore include the Go run-time, along with the run-time type information necessary to support dynamic type checks, reflection, and even panic-time stack traces.

A simple C "hello, world" program compiled and linked statically using `gcc` on

Linux is around 750 kB, including an implementation of `printf`. An equivalent Go program using `fmt.Printf` is around 1.2 MB, but that includes more powerful run-time support.

Can I stop these complaints about my unused variable/import?

The presence of an unused variable may indicate a bug, while unused imports just slow down compilation. Accumulate enough unused imports in your code tree and things can get very slow. For these reasons, Go allows neither.

When developing code, it's common to create these situations temporarily and it can be annoying to have to edit them out before the program will compile.

Some have asked for a compiler option to turn those checks off or at least reduce them to warnings. Such an option has not been added, though, because compiler options should not affect the semantics of the language and because the Go compiler does not report warnings, only errors that prevent compilation.

There are two reasons for having no warnings. First, if it's worth complaining about, it's worth fixing in the code. (And if it's not worth fixing, it's not worth mentioning.) Second, having the compiler generate warnings encourages the implementation to warn about weak cases that can make compilation noisy, masking real errors that *should* be fixed.

It's easy to address the situation, though. Use the blank identifier to let unused things persist while you're developing.

```
import "unused"

// This declaration marks the import as used by referencing an
// item from the package.
var _ = unused.Item // TODO: Delete before committing!

func main() {
    debugData := debug.Profile()
    _ = debugData // Used only during debugging.
    ....
}
```

Performance

Why does Go perform badly on benchmark X?

One of Go's design goals is to approach the performance of C for comparable programs, yet on some benchmarks it does quite poorly, including several in [test/bench/shootout](#). The slowest depend on libraries for which versions of comparable performance are not available in Go. For instance, [pidigits.go](#) depends on a multi-precision math package, and the C versions, unlike Go's, use [GMP](#) (which is written in optimized assembler). Benchmarks that depend on regular expressions ([regex-dna.go](#), for instance) are essentially comparing Go's native [regexp package](#) to mature, highly optimized regular expression libraries like PCRE.

Benchmark games are won by extensive tuning and the Go versions of most of the benchmarks need attention. If you measure comparable C and Go programs ([reverse-complement.go](#) is one example), you'll see the two languages are much closer in raw performance than this suite would indicate.

Still, there is room for improvement. The compilers are good but could be better, many libraries need major performance work, and the garbage collector isn't fast enough yet. (Even if it were, taking care not to generate unnecessary garbage can have a huge effect.)

In any case, Go can often be very competitive. There has been significant improvement in the performance of many programs as the language and tools have developed. See the blog post about [profiling Go programs](#) for an informative example.

Changes from C

Why is the syntax so different from C?

Other than declaration syntax, the differences are not major and stem from two desires. First, the syntax should feel light, without too many mandatory keywords, repetition, or arcana. Second, the language has been designed to be easy to analyze and can be parsed without a symbol table. This makes it much easier to build tools such as debuggers, dependency analyzers, automated documentation extractors, IDE plug-ins, and so on. C and its descendants are notoriously difficult in this regard.

Why are declarations backwards?

They're only backwards if you're used to C. In C, the notion is that a variable is declared like an expression denoting its type, which is a nice idea, but the type and expression grammars don't mix very well and the results can be confusing; consider function pointers. Go mostly separates expression and type syntax and that simplifies things (using prefix `*` for pointers is an exception that proves the rule). In C, the declaration

```
int* a, b;
```

declares `a` to be a pointer but not `b`; in Go

```
var a, b *int
```

declares both to be pointers. This is clearer and more regular. Also, the `:=` short declaration form argues that a full variable declaration should present the same order as `:=` so

```
var a uint64 = 1
```

has the same effect as

```
a := uint64(1)
```

Parsing is also simplified by having a distinct grammar for types that is not just the expression grammar; keywords such as `func` and `chan` keep things clear.

See the article about [Go's Declaration Syntax](#) for more details.

Why is there no pointer arithmetic?

Safety. Without pointer arithmetic it's possible to create a language that can never derive an illegal address that succeeds incorrectly. Compiler and hardware technology have advanced to the point where a loop using array indices can be as efficient as a loop using pointer arithmetic. Also, the lack of pointer arithmetic can simplify the implementation of the garbage collector.

Why are ++ and -- statements and not expressions? And why postfix, not prefix?

Without pointer arithmetic, the convenience value of pre- and postfix increment operators drops. By removing them from the expression hierarchy altogether, expression syntax is simplified and the messy issues around order of evaluation of ++ and -- (consider $f(i++)$ and $p[i] = q[++i]$) are eliminated as well. The simplification is significant. As for postfix vs. prefix, either would work fine but the postfix version is more traditional; insistence on prefix arose with the STL, a library for a language whose name contains, ironically, a postfix increment.

Why are there braces but no semicolons? And why can't I put the opening brace on the next line?

Go uses brace brackets for statement grouping, a syntax familiar to programmers who have worked with any language in the C family. Semicolons, however, are for parsers, not for people, and we wanted to eliminate them as much as possible. To achieve this goal, Go borrows a trick from BCPL: the semicolons that separate statements are in the formal grammar but are injected automatically, without lookahead, by the lexer at the end of any line that could be the end of a statement. This works very well in practice but has the effect that it forces a brace style. For instance, the opening brace of a function cannot appear on a line by itself.

Some have argued that the lexer should do lookahead to permit the brace to live on the next line. We disagree. Since Go code is meant to be formatted automatically by [gofmt](#), *some* style must be chosen. That style may differ from what you've used in C or Java, but Go is a new language and gofmt's style is as

good as any other. More important—much more important—the advantages of a single, programmatically mandated format for all Go programs greatly outweigh any perceived disadvantages of the particular style. Note too that Go's style means that an interactive implementation of Go can use the standard syntax one line at a time without special rules.

Why do garbage collection? Won't it be too expensive?

One of the biggest sources of bookkeeping in systems programs is memory management. We feel it's critical to eliminate that programmer overhead, and advances in garbage collection technology in the last few years give us confidence that we can implement it with low enough overhead and no significant latency.

Another point is that a large part of the difficulty of concurrent and multi-threaded programming is memory management; as objects get passed among threads it becomes cumbersome to guarantee they become freed safely. Automatic garbage collection makes concurrent code far easier to write. Of course, implementing garbage collection in a concurrent environment is itself a challenge, but meeting it once rather than in every program helps everyone.

Finally, concurrency aside, garbage collection makes interfaces simpler because they don't need to specify how memory is managed across them.

The current implementation is a parallel mark-and-sweep collector but a future version might take a different approach.

On the topic of performance, keep in mind that Go gives the programmer considerable control over memory layout and allocation, much more than is typical in garbage-collected languages. A careful programmer can reduce the garbage collection overhead dramatically by using the language well; see the article about [profiling Go programs](#) for a worked example, including a demonstration of Go's profiling tools.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

[Project Home Wiki](#)

Search for

Search

★ **GOPATH**

The GOPATH environment variable and its uses

Updated Apr 6, 2012 by [kev...@google.com](#)

Introduction

The GOPATH environment variable is used to specify directories outside of \$GOROOT that contain the source for Go projects and their binaries.

GOPATH is used by goinstall and the "go" tool as a destination for the binaries it builds and a location to search for imports.

GOPATH is a path list - multiple directories can be specified by separating them with a ":" (on os x or linux) or a ";" (on windows). When multiple directories are listed, and goinstall or "go" are used from outside any of them, the first directory is used as the installation destination. When using either tool from within one of the listed directories, the containing directory is used as the installation destination.

For most of this document, \$GOPATH will refer to whichever of the listed directories is the currently active one.

Integrating GOPATH

On os x or linux, adding the following expression to PATH will add all \$GOPATH

```
${GOPATH//:/:bin:}/bin
```

Adding the following block to a standard Go makefile will bring in all \$GOPATH

```
GOPATHSEP=:
ifeq ($(GOHOSTOS),windows)
GOPATHSEP=;
endif
GCIMPORTS+=-I $(subst $(GOPATHSEP),/pkg/$(GOOS)_$(GOARCH) -I , $(GO
LDIMPORTS+=-L $(subst $(GOPATHSEP),/pkg/$(GOOS)_$(GOARCH) -L , $(GO
```

goinstall and the "go" tool already know about GOPATH.

Directory layout

The source for a package with the import path "X/Y/Z" is in the directory

`$GOPATH/src/X/Y/Z`

The binary for a package with the import path "X/Y/Z" is in

`$GOPATH/pkg/$GOOS_$GOARCH/X/Y/Z.a`

The binary for a command whose source is in `$GOPATH/src/A/B` is

`$GOPATH/bin/B`

repository integration and creating "goinstallable" projects

goinstall, when fetching a package, looks at the package's import path to discover a URL. For instance, if you attempt to

```
goinstall code.google.com/p/gomatrix/matrix
```

goinstall will get the source from the project hosted at <http://code.google.com/p/gomatrix>, and it will clone the repository to

```
$GOPATH/src/code.google.com/p/gomatrix
```

As a result, if (from your repository project) you import a package that is in the same repository, you need to use its "full" import path - the place goinstall puts it. In this example, if something else wants to import the "matrix" package, it should import "code.google.com/p/gomatrix/matrix" rather than "matrix".

If you prefer to use makefiles to build on your own machine and you still want your project to work well with goinstall, set the TARG variable to the long import path. goinstall will ignore this makefile, but as long as TARG matches the package's location relative to the repository, goinstall will choose the same import path.

Tips and tricks

Third-party Packages

It is useful to have two GOPATH entries. One for a location for 3rd party goinstalled packages, and the second for your own projects. List the 3rd party GOPATH first, so that goinstall will use it as a default destination. Then you can work in the second GOPATH directory and have all your packages be importable by using the "go" command, goinstall, or a GOPATH-aware 3rd party build tool like [gb](#).

FAQ

Why won't `$GOPATH/src/cmd/mycmd/*.go` build?

When the go command is looking for packages, it always looks in `$GOROOT` first. This includes directories, so if it finds (as in the case above) a `cmd/` directory in `$GOROOT` it won't proceed to look in any of the `GOPATH` directories. This prevents you from defining your own `math/matrix` package as well as your own `cmd/mycmd` commands.

[Project Home Wiki](#)

Search for

Search

★ **SQLDrivers**

SQL database drivers [database](#), [sql](#)

Updated Apr 5, 2012 by bradfitz@golang.org

SQL database drivers

The database/sql and database/sql/driver packages are designed for using databases from Go and implementing database drivers, respectively.

See the design goals doc:

<http://golang.org/src/pkg/database/sql/doc.txt>

Drivers

Drivers for Go's sql package include:

- **MS ADODB:** <https://github.com/mattn/go-adodb>
- **MySQL:** <https://github.com/ziutek/mymysql>
- **ODBC:** <https://bitbucket.org/miquella/mgodbc>
- **Postgres** (uses cgo): <https://github.com/jbarham/gopgsqldriver>
- **Postgres** (pure Go): <https://github.com/bmizerany/pq>
- **SQLite:** <https://github.com/mattn/go-sqlite3>
- **Oracle:** <https://github.com/mattn/go-oci8>
- **DB2:** <https://bitbucket.org/phiggins/go-db2-cli>

[Project Home Wiki](#)

Search for

Search

★ **WindowsDLLs**

calling Windows DLLs from Go [windows](#), [syscall](#), [dll](#), [sample](#)

Updated May 23, 2011 by [a...@golang.org](#)

Calling a Windows DLL

A sample program that calls Windows DLLs from Go:

```
package main

import (
    "syscall"
    "unsafe"
    "fmt"
)

func abort(funcname string, err int) {
    panic(funcname + " failed: " + syscall.Errstr(err))
}

var (
    kernel32, _ = syscall.LoadLibrary("kernel32.dll")
    getModuleHandle, _ = syscall.GetProcAddress(kernel32, "GetMo

    user32, _ = syscall.LoadLibrary("user32.dll")
    messageBox, _ = syscall.GetProcAddress(user32, "MessageBoxW"
)

const (
    MB_OK                = 0x00000000
    MB_OKCANCEL          = 0x00000001
    MB_ABORTRETRYIGNORE = 0x00000002
    MB_YESNOCANCEL       = 0x00000003
    MB_YESNO             = 0x00000004
    MB_RETRYCANCEL       = 0x00000005
    MB_CANCELTRYCONTINUE = 0x00000006
    MB_ICONHAND          = 0x00000010
    MB_ICONQUESTION      = 0x00000020
    MB_ICONEXCLAMATION   = 0x00000030
    MB_ICONASTERISK      = 0x00000040
    MB_USERICON          = 0x00000080
    MB_ICONWARNING       = MB_ICONEXCLAMATION
    MB_ICONERROR         = MB_ICONHAND
    MB_ICONINFORMATION   = MB_ICONASTERISK
    MB_ICONSTOP          = MB_ICONHAND

    MB_DEFBUTTON1        = 0x00000000
```

```

        MB_DEFBUTTON2           = 0x00000100
        MB_DEFBUTTON3           = 0x00000200
        MB_DEFBUTTON4           = 0x00000300
    )

func MessageBox(caption, text string, style uintptr) (result int) {
    ret, _, callErr := syscall.Syscall9(uintptr(messageBox),
        0,
        uintptr(unsafe.Pointer(syscall.StringToUTF16Ptr(text))),
        uintptr(unsafe.Pointer(syscall.StringToUTF16Ptr(caption))),
        style,
        0,
        0,
        0,
        0,
        0)
    if callErr != 0 {
        abort("Call MessageBox", int(callErr))
    }
    result = int(ret)
    return
}

func GetModuleHandle() (handle uint) {
    if ret, _, callErr := syscall.Syscall(uintptr(getModuleHandle),
        0,
        abort("Call GetModuleHandle", int(callErr))
    } else {
        handle = ret
    }
    return
}

func main() {
    defer syscall.FreeLibrary(kernel32)
    defer syscall.FreeLibrary(user32)

    fmt.Printf("Return: %d\n", MessageBox("Done Title", "This te
}

func init() {
    fmt.Print("Starting Up\n")
}

```

[Project Home Wiki](#)

Search for

☆ **GoForCPPProgrammers**

Go for C++ Programmers

Updated Mar 6, 2012 by a...@golang.org

Go is a systems programming language intended to be a general-purpose systems language, like C++. These are some notes on Go for experienced C++ programmers. This document discusses the differences between Go and C++, and says little to nothing about the similarities.

For a more general introduction to Go, see the [Go Tour](#), [How to Write Go Code](#) and [Effective Go](#).

For a detailed description of the Go language, see the [Go spec](#).

Conceptual Differences

- Go does not have classes with constructors or destructors. Instead of class methods, a class inheritance hierarchy, and virtual functions, Go provides *interfaces*, which are discussed in more detail below. Interfaces are also used where C++ uses templates.
- Go uses garbage collection. It is not necessary (or possible) to release memory explicitly.
- Go has pointers but not pointer arithmetic. You cannot use a pointer variable to walk through the bytes of a string.
- Arrays in Go are first class values. When an array is used as a function parameter, the function receives a copy of the array, not a pointer to it. However, in practice functions often use slices for parameters; slices hold pointers to underlying arrays. Slices are discussed further below.
- Strings are provided by the language. They may not be changed once they have been created.
- Hash tables are provided by the language. They are called maps.
- Separate threads of execution, and communication channels between them, are provided by the language. This is discussed further below.
- Certain types (maps and channels, described further below) are passed by reference, not by value. That is, passing a map to a function does not copy the map, and if the function changes the map the change will be seen by the caller. In C++ terms, one can think of these as being reference types.
- Go does not use header files. Instead, each source file is part of a defined *package*. When a package defines an object (type, constant, variable, function) with a name starting with an upper case letter, that object is visible to any other file which imports that package.
- Go does not support implicit type conversion. Operations that mix different types require casts (called conversions in Go).

- Go does not support function overloading and does not support user defined operators.
- Go does not support `const` or `volatile` qualifiers.
- Go uses `nil` for invalid pointers, where C++ uses `NULL` or simply `0`.

Syntax

The declaration syntax is reversed compared to C++. You write the name followed by the type. Unlike in C++, the syntax for a type does not match the way in which the variable is used. Type declarations may be read easily from left to right.

```
Go                C++
var v1 int        // int v1;
var v2 string     // const std::string v2; (approximately)
var v3 [10]int    // int v3[10];
var v4 []int      // int* v4; (approximately)
var v5 struct { f int } // struct { int f; } v5;
var v6 *int       // int* v6; (but no pointer arithmetic)
var v7 map[string]int // unordered_map<string, int>* v7; (approx)
var v8 func(a int) int // int (*v8)(int a);
```

Declarations generally take the form of a keyword followed by the name of the variable being declared. The keyword is one of `var`, `func`, `const`, or `type`. Method declarations are an exception in that the receiver appears before the name of the object being declared. See the discussion of interfaces.

You can also use a keyword followed by a series of declarations in parentheses.

```
var (
    i int
    m float64
)
```

When declaring a function, you must either provide a name for each parameter or a name for any parameter; you can't omit some names and provide others. You may use several names with the same type:

```
func f(i, j, k int, s, t string)
```

A variable may be initialized when it is declared. When this is done, specifying the type is permitted but not required. When the type is not specified, the type of the variable is the type of the initialization expression.

```
var v = *p
```

See also the discussion of constants, below. If a variable is not initialized explicitly

must be specified. In that case it will be implicitly initialized to the type's zero value (e.g., 0 for integers, false for booleans, etc.). There are no uninitialized variables in Go.

Within a function, a short declaration syntax is available with `:=`.

```
v1 := v2
```

This is equivalent to

```
var v1 = v2
```

Go permits multiple assignments, which are done in parallel.

```
i, j = j, i // Swap i and j.
```

Functions may have multiple return values, indicated by a list in parentheses. The values can be stored by assignment to a list of variables.

```
func f() (i int, j int) { ... }  
v1, v2 = f()
```

Go code uses very few semicolons in practice. Technically, all Go statements are terminated by a semicolon. However, Go treats the end of a non-blank line as a semicolon unless the line is clearly incomplete (the exact rules are in the language specification). A consequence is that in some cases Go does not permit you to use a line break. For example, you may not write

```
func g()  
{  
    // INVALID  
}
```

A semicolon will be inserted after `g()`, causing it to be a function declaration rather than a function definition. Similarly, you may not write

```
if x {  
}  
else {  
    // INVALID  
}
```

A semicolon will be inserted after the `}` preceding the `else`, causing a syntax error.

Since semicolons do not end statements, you may continue using them as in C++. However, this is not the recommended style. Idiomatic Go code omits unnecessary semicolons,

practice is all of them other than the initial for loop clause and cases where you short statements on a single line.

While we're on the topic, we recommend that rather than worry about semicolon placement, you format your code with the `gofmt` program. That will produce a standard Go style, and let you worry about your code rather than your formatting. While the rules initially seem odd, it is as good as any other style, and familiarity will lead to comfort.

When using a pointer to a struct, you use `.` instead of `->`. Thus syntactically speaking, a struct and a pointer to a structure are used in the same way.

```
type myStruct struct { i int }
var v9 myStruct           // v9 has structure type
var p9 *myStruct         // p9 is a pointer to a structure
f(v9.i, p9.i)
```

Go does not require parentheses around the condition of an `if` statement, or the condition of a `for` statement, or the value of a `switch` statement. On the other hand, it does require braces around the body of an `if` or `for` statement.

```
if a < b { f() }           // Valid
if (a < b) { f() }        // Valid (condition is a parenthesized
if (a < b) f()            // INVALID
for i = 0; i < 10; i++ {} // Valid
for (i = 0; i < 10; i++) {} // INVALID
```

Go does not have a `while` statement nor does it have a `do/while` statement. The `for` statement may be used with a single condition, which makes it equivalent to a `while` statement. Omitting the condition entirely is an endless loop.

Go permits `break` and `continue` to specify a label. The label must refer to a `for`, `select` statement.

In a `switch` statement, case labels do not fall through. You can make them fall through with the `fallthrough` keyword. This applies even to adjacent cases.

```
switch i {
case 0: // empty case body
case 1:
    f() // f is not called when i == 0!
}
```

But a case can have multiple values.

```
switch i {
case 0, 1:
    f() // f is called if i == 0 || i == 1.
}
```

The values in a case need not be constants--or even integers; any type that supports equality comparison operator, such as strings or pointers, can be used--and if the value is omitted it defaults to `true`.

```
switch {
case i < 0:
    f1()
case i == 0:
    f2()
case i > 0:
    f3()
}
```

The `++` and `--` operators may only be used in statements, not in expressions. You can write `c = *p++`. `*p++` is parsed as `(*p)++`.

The `defer` statement may be used to call a function after the function containing the statement returns.

```
fd := open("filename")
defer close(fd) // fd will be closed when this function returns
```

Constants

In Go constants may be *untyped*. This applies even to constants named with a `const` declaration, if no type is given in the declaration and the initializer expression uses only untyped constants. A value derived from an untyped constant becomes typed when it is used within a context that requires a typed value. This permits constants to be used relatively freely without requiring general implicit type conversion.

```
var a uint
f(a + 1) // untyped numeric constant "1" becomes typed as uint
```

The language does not impose any limits on the size of an untyped numeric constant or constant expression. A limit is only applied when a constant is used where a type is required.

```
const huge = 1 << 100
f(huge >> 98)
```

Go does not support enums. Instead, you can use the special name `iota` in a single `const` declaration to get a series of increasing value. When an initialization expression is omitted for a `const`, it reuses the preceding expression.

```
const (
    red = iota // red == 0
    blue      // blue == 1
    green     // green == 2
)
```

Slices

A slice is conceptually a struct with three fields: a pointer to an array, a length, and a capacity. Slices support the `[]` operator to access elements of the underlying array. The builtin `len` function returns the length of the slice. The builtin `cap` function returns the capacity.

Given an array, or another slice, a new slice is created via `a[i:j]`. This creates a new slice which refers to `a`, starts at index `i`, and ends before index `j`. It has length `j - i`. If `i` is omitted, the slice starts at `0`. If `j` is omitted, the slice ends at `len(a)`. The new slice refers to the same array to which `a` refers. That is, changes made using the new slice may be seen using `a`. The capacity of the new slice is simply the capacity of `a` minus `i`. The capacity of an array is the length of the array.

What this means is that Go uses slices for some cases where C++ uses pointers. If you create a value of type `[100]byte` (an array of 100 bytes, perhaps a buffer) and you want to pass it to a function without copying it, you should declare the function parameter to have type `[]byte`, and pass a slice of the array (`a[:]` will pass the entire array). Unlike in C++, it is not necessary to pass the length of the buffer; it is efficiently accessible via `len`.

The slice syntax may also be used with a string. It returns a new string, whose value is a substring of the original string. Because strings are immutable, string slices can be implemented without allocating new storage for the slices's contents.

Making values

Go has a builtin function `new` which takes a type and allocates space on the heap. The allocated space will be zero-initialized for the type. For example, `new(int)` allocates a new `int` on the heap, initializes it with the value `0`, and returns its address, which has type `*int`. Unlike in C++, `new` is a function, not an operator; `new int` is a syntax error.

Perhaps surprisingly, `new` is not commonly used in Go programs. In Go taking the address of a variable is always safe and never yields a dangling pointer. If the program takes the address of a variable, it will be allocated on the heap if necessary. So these functions are equivalent:

```
type S { I int }

func f1() *S {
    return new(S)
}

func f2() *S {
    var s S
    return &s
}

func f3() *S {
    // More idiomatic: use composite literal syntax.
    return &S{0}
}
```

Map and channel values must be allocated using the builtin function `make`. A variable declared with map or channel type without an initializer will be automatically initialized to `nil`. Calling `make(map[int]int)` returns a newly allocated value of type `map[int]int`. Note that `make` returns a value, not a pointer. This is consistent with the fact that map and channel values are passed by reference. Calling `make` with a map type takes an optional argument which is the expected capacity of the map. Calling `make` with a channel type takes an optional argument which sets the buffering capacity of the channel; the default is 0 (unbuffered).

The `make` function may also be used to allocate a slice. In this case it allocates memory for the underlying array and returns a slice referring to it. There is one

required argument, which is the number of elements in the slice. A second, optional, argument is the capacity of the slice. For example, `make([]int, 10, 20)`. This is identical to `new([20]int)[0:10]`. Since Go uses garbage collection, the newly allocated array will be discarded sometime after there are no references to the returned slice.

Interfaces

Where C++ provides classes, subclasses and templates, Go provides interfaces. An interface is similar to a C++ pure abstract class: a class with no data members, with methods which are all pure virtual. However, in Go, any type which provides the methods named in the interface may be treated as an implementation of the interface. No declared inheritance is required. The implementation of the interface is entirely separate from the interface itself.

A method looks like an ordinary function definition, except that it has a *receiver*. The receiver is similar to the `this` pointer in a C++ class method.

```
type myType struct { i int }
func (p *myType) Get() int { return p.i }
```

This declares a method `Get` associated with `myType`. The receiver is named `p` in the function.

Methods are defined on named types. If you convert the value to a different type, the value will have the methods of the new type, not the old type.

You may define methods on a builtin type by declaring a new named type derived from the builtin type. The new type is distinct from the builtin type.

```
type myInteger int
func (p myInteger) Get() int { return int(p) } // Conversion required
func f(i int) { }
var v myInteger
// f(v) is invalid.
// f(int(v)) is valid; int(v) has no defined methods.
```

Given this interface:

```
type myInterface interface {
    Get() int
    Set(i int)
}
```

we can make `myType` satisfy the interface by adding

```
func (p *myType) Set(i int) { p.i = i }
```

Now any function which takes `myInterface` as a parameter will accept a variable `*myType`.

```
func GetAndSet(x myInterface) {}  
func f1() {  
    var p myType  
    GetAndSet(&p)  
}
```

In other words, if we view `myInterface` as a C++ pure abstract base class, defining `Get` and `Set` for `*myType` made `*myType` automatically inherit from `myInterface`. A type can satisfy multiple interfaces.

An anonymous field may be used to implement something much like a C++ child class.

```
type myChildType struct { myType; j int }  
func (p *myChildType) Get() int { p.j++; return p.myType.Get() }
```

This effectively implements `myChildType` as a child of `myType`.

```
func f2() {  
    var p myChildType  
    GetAndSet(&p)  
}
```

The `Set` method is effectively inherited from `myType`, because methods associated with the anonymous field are promoted to become methods of the enclosing type. In this example, because `myChildType` has an anonymous field of type `myType`, the methods of `myType` also become methods of `myChildType`. In this example, the `Get` method was overridden and the `Set` method was inherited.

This is not precisely the same as a child class in C++. When a method of an anonymous field is called, its receiver is the field, not the surrounding struct. In other words, anonymous fields are not virtual functions. When you want the equivalent of a child class, use an interface.

A variable that has an interface type may be converted to have a different interface using a special construct called a type assertion. This is implemented dynamically at runtime, like C++ `dynamic_cast`. Unlike `dynamic_cast`, there does not need to be a declared relationship between the two interfaces.

```
type myPrintInterface interface {  
    Print()  
}
```

```

}
func f3(x myInterface) {
    x.(myPrintInterface).Print() // type assertion to myPrintI
}

```

The conversion to `myPrintInterface` is entirely dynamic. It will work as long as the underlying type of `x` (the *dynamic type*) defines a `Print` method.

Because the conversion is dynamic, it may be used to implement generic programs similar to templates in C++. This is done by manipulating values of the minimal interface.

```

type Any interface { }

```

Containers may be written in terms of `Any`, but the caller must unbox using a type assertion to recover values of the contained type. As the typing is dynamic rather than static, there is no equivalent of the way that a C++ template may inline the relevant operations. The operations are fully type-checked at run time, but all operations involve a function call.

```

type Iterator interface {
    Get() Any
    Set(v Any)
    Increment()
    Equal(arg Iterator) bool
}

```

Note that `Equal` has an argument of type `Iterator`. This does not behave like a C++ template. See the FAQ.

Goroutines

Go permits starting a new thread of execution (a *goroutine*) using the `go` statement. The `go` statement runs a function in a different, newly created, goroutine. All goroutines in a single program share the same address space.

Internally, goroutines act like coroutines that are multiplexed among multiple operating system threads. You do not have to worry about these details.

```
func server(i int) {
    for {
        fmt.Print(i)
        time.Sleep(10 * time.Second)
    }
}
go server(1)
go server(2)
```

(Note that the `for` statement in the `server` function is equivalent to a C++ `while (true)` loop.)

Goroutines are (intended to be) cheap.

Function literals (which Go implements as closures) can be useful with the `go` statement.

```
var g int
go func(i int) {
    s := 0
    for j := 0; j < i; j++ { s += j }
    g = s
}(1000) // Passes argument 1000 to the function literal.
```

Channels

Channels are used to communicate between goroutines. Any value may be sent over a channel. Channels are (intended to be) efficient and cheap. To send a value on a channel, use `<-` as a binary operator. To receive a value on a channel, use `<-` as a unary operator. When calling functions, channels are passed by reference.

The Go library provides mutexes, but you can also use a single goroutine with a shared channel. Here is an example of using a manager function to control access to a single value.

```
type Cmd struct { Get bool; Val int }
func Manager(ch chan Cmd) {
    val := 0
    for {
        c := <-ch
        if c.Get { c.Val = val; ch <- c }
        else { val = c.Val }
    }
}
```

In that example the same channel is used for input and output. This is incorrect if there are multiple goroutines communicating with the manager at once: a goroutine waiting for a response from the manager might receive a request from another goroutine instead. A solution is to pass in a channel.

```
type Cmd2 struct { Get bool; Val int; Ch <- chan int }
func Manager2(ch chan Cmd2) {
    val := 0
    for {
        c := <-ch
        if c.Get { c.Ch <- val }
        else { val = c.Val }
    }
}
```

To use `Manager2`, given a channel to it:

```
func f4(ch <- chan Cmd2) int {
    myCh := make(chan int)
    c := Cmd2{ true, 0, myCh } // Composite literal syntax.
    ch <- c
}
```

```
    return <-myCh  
}
```

Powered by [Google Project Hosting Help](#)

[Project Home Wiki](#)

Search for

Search

★ **cgo**

Tips for interfacing with C code and libraries using cgo.

Updated Mar 30, 2012 by minux...@gmail.com

Introduction

First, <http://golang.org/cmd/cgo> is the primary cgo documentation.

There is also a good introduction article at http://golang.org/doc/articles/c_go_cgo.html.

The basics

If a Go source file imports "C", it is using cgo. The Go file will have access to anything appearing in the comment immediately preceding the line `import "C"`, and will be linked against all other cgo comments in other Go files, and all C files included in the build process.

Note that there must be no blank lines in between the cgo comment and the `import` statement.

To access a symbol originating from the C side, use the package name `c`. That is, if you want to call the C function `printf()` from Go code, you write `C.printf()`.

```
package cgoexample

/*
#include <stdio.h>
#include <stdlib.h>

void myprint(char* s) {
    printf("%s", s);
}
*/
import "C"

import "unsafe"

func Example() {
    cs := C.CString("Hello from stdio\n")
    C.myprint(cs)
    C.free(unsafe.Pointer(cs))
}
```

Calling Go functions from C

It is possible to call both top-level Go functions and function variables from C code.

Global functions

Go makes its functions available to C code through use of a special `//export c` directive.

```
package gocallback

import "fmt"

/*
#include <stdio.h>

extern void AGoFunction();

void ACFunction() {
    printf("ACFunction()\n");
    AGoFunction();
}
*/
import "C"

//export AGoFunction
func AGoFunction() {
    fmt.Println("AGoFunction()")
}

func Example() {
    C.ACFunction()
}
```

Function variables

The following code shows an example of invoking a Go callback from C code. `CallMyFunction()`. `CallMyFunction()` invokes the callback by sending it back to the Go runtime.

```
package gocallback

import (
    "unsafe"
    "fmt"
)
```

```
/*
extern void go_callback_int(void* foo, int p1);

void CallMyFunction(void* pfoo) {
    go_callback_int(pfoo, 5);
}
*/
import "C"

//export go_callback_int
func go_callback_int(pfoo unsafe.Pointer, p1 C.int) {
    foo := *(*func(C.int))(pfoo)
    foo(p1)
}

func MyCallback(x C.int) {
    fmt.Println("callback with", x)
}
//we store it in a global variable so that the garbage collector do
var MyCallbackFunc = MyCallback

func Example() {
    C.CallMyFunction(unsafe.Pointer(&MyCallbackFunc))
}
```

Go strings and C strings

Go strings and C strings are different. Go strings are the combination of a length and a pointer to the first character in the string. C strings are just the pointer to the first character, and are terminated by the first instance of the null character, `'\0'`.

Go provides means to go from one to another in the form of the following three functions:

- `func C.CString(goString string) *C.char`
- `func C.GoString(cString *C.char) string`
- `func C.GoStringN(cString *C.char, length C.int) string`

One important thing to remember is that `C.CString()` will allocate a new string of the appropriate length, and return it. That means the C string is not going to be garbage collected and it is up to **you** to free it. A standard way to do this follows.

```
// #include <stdlib.h>
import "C"
import "unsafe"
...
    var cmsg *C.char = C.CString("hi")
    defer C.free(unsafe.Pointer(cmsg))
    // do something with the C string
```

Of course, you aren't required to use `defer` to call `C.free()`. You can free the C string whenever you like, but it is your responsibility to make sure it happens.

Turning C arrays into Go slices

C arrays are typically either null-terminated or have a length kept elsewhere.

Go provides the following function to make a new Go byte slice from a C array:

- `func C.GoBytes(cArray unsafe.Pointer, length C.int) []byte`

To create a Go slice backed by a C array (without copying the original data), one acquires this length at runtime and uses `reflect.SliceHeader`.

```
import "C"
import "unsafe"
...
var theCArray *TheCType := C.getTheArray()
length := C.getTheArrayLength()
var theGoSlice []TheCType
sliceHeader := (*reflect.SliceHeader)((unsafe.Pointer(&theGoSlice))
sliceHeader.Cap = length
sliceHeader.Len = length
sliceHeader.Data = uintptr(unsafe.Pointer(&theCArray[0]))
// now theGoSlice is a normal Go slice backed by the C array
```

It is important to keep in mind that the Go garbage collector will not interact with C code and that if it is freed from the C side of things, the behavior of any Go code using the slice is nondeterministic.

Common Pitfalls

Struct Alignment Issues

As Go doesn't support packed struct (e.g., structs where maximum alignment is 1 byte), you can't use packed C struct in Go. Even if your program passes compilation, it won't do what you want. To use it, you have to read/write the struct as byte array/slice.

//export and definition in preamble

If your program uses any `//export` directives, then the C code in the comment may only include declarations (`extern int f();`), not definitions (`int f() { return 1; }` or `int n;`).

Writing Web Applications

Introduction

Covered in this tutorial:

- Creating a data structure with load and save methods
- Using the `net/http` package to build web applications
- Using the `html/template` package to process HTML templates
- Using the `regexp` package to validate user input
- Using closures

Assumed knowledge:

- Programming experience
- Understanding of basic web technologies (HTTP, HTML)
- Some UNIX/DOS command-line knowledge

Getting Started

At present, you need to have a FreeBSD, Linux, OS X, or Windows machine to run Go. We will use \$ to represent the command prompt.

Install Go (see the [Installation Instructions](#)).

Make a new directory for this tutorial inside your GOPATH and cd to it:

```
$ mkdir gowiki
$ cd gowiki
```

Create a file named `wiki.go`, open it in your favorite editor, and add the following lines:

```
package main

import (
    "fmt"
    "io/ioutil"
)
```

We import the `fmt` and `ioutil` packages from the Go standard library. Later, as we implement additional functionality, we will add more packages to this `import` declaration.

Data Structures

Let's start by defining the data structures. A wiki consists of a series of interconnected pages, each of which has a title and a body (the page content). Here, we define `Page` as a struct with two fields representing the title and body.

```
type Page struct {
    Title string
    Body  []byte
}
```

The type `[]byte` means "a byte slice". (See [Slices: usage and internals](#) for more on slices.) The `Body` element is a `[]byte` rather than `string` because that is the type expected by the `io` libraries we will use, as you'll see below.

The `Page` struct describes how page data will be stored in memory. But what about persistent storage? We can address that by creating a `save` method on `Page`:

```
func (p *Page) save() error {
    filename := p.Title + ".txt"
    return ioutil.WriteFile(filename, p.Body, 0600)
}
```

This method's signature reads: "This is a method named `save` that takes as its receiver `p`, a pointer to `Page`. It takes no parameters, and returns a value of type `error`."

This method will save the `Page`'s `Body` to a text file. For simplicity, we will use the `Title` as the file name.

The `save` method returns an error value because that is the return type of `writeFile` (a standard library function that writes a byte slice to a file). The `save` method returns the error value, to let the application handle it should anything go wrong while writing the file. If all goes well, `Page.save()` will return `nil` (the zero-value for pointers, interfaces, and some other types).

The octal integer constant `0600`, passed as the third parameter to `writeFile`, indicates that the file should be created with read-write permissions for the current user only. (See the Unix man page `open(2)` for details.)

We will want to load pages, too:

```
func loadPage(title string) *Page {
    filename := title + ".txt"
    body, _ := ioutil.ReadFile(filename)
    return &Page{Title: title, Body: body}
}
```

The function `loadPage` constructs the file name from `title`, reads the file's contents into a new `Page`, and returns a pointer to that new page.

Functions can return multiple values. The standard library function `io.ReadFile` returns `[]byte` and `error`. In `loadPage`, error isn't being handled yet; the "blank identifier" represented by the underscore (`_`) symbol is used to throw away the error return value (in essence, assigning the value to nothing).

But what happens if `ReadFile` encounters an error? For example, the file might not exist. We should not ignore such errors. Let's modify the function to return `*Page` and `error`.

```
func loadPage(title string) (*Page, error) {
    filename := title + ".txt"
    body, err := ioutil.ReadFile(filename)
    if err != nil {
        return nil, err
    }
    return &Page{Title: title, Body: body}, nil
}
```

Callers of this function can now check the second parameter; if it is `nil` then it has successfully loaded a `Page`. If not, it will be an error that can be handled by the caller (see the [language specification](#) for details).

At this point we have a simple data structure and the ability to save to and load from a file. Let's write a main function to test what we've written:

```
func main() {
    p1 := &Page{Title: "TestPage", Body: []byte("This is a sample Pa
p1.save()
    p2, _ := loadPage("TestPage")
    fmt.Println(string(p2.Body))
}
```

After compiling and executing this code, a file named `TestPage.txt` would be

created, containing the contents of p1. The file would then be read into the struct p2, and its Body element printed to the screen.

You can compile and run the program like this:

```
$ go build wiki.go
$ ./wiki
This is a sample page.
```

(If you're using Windows you must type "wiki" without the "./" to run the program.)

[Click here to view the code we've written so far.](#)

Introducing the `net/http` package (an interlude)

Here's a full working example of a simple web server:

```
package main

import (
    "fmt"
    "net/http"
)

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hi there, I love %s!", r.URL.Path[1:])
}

func main() {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

The `main` function begins with a call to `http.HandleFunc`, which tells the `http` package to handle all requests to the web root (`/`) with `handler`.

It then calls `http.ListenAndServe`, specifying that it should listen on port 8080 on any interface (`:8080`). (Don't worry about its second parameter, `nil`, for now.) This function will block until the program is terminated.

The function `handler` is of the type `http.HandlerFunc`. It takes an `http.ResponseWriter` and an `http.Request` as its arguments.

An `http.ResponseWriter` value assembles the HTTP server's response; by writing to it, we send data to the HTTP client.

An `http.Request` is a data structure that represents the client HTTP request. The string `r.URL.Path` is the path component of the request URL. The trailing `[1:]` means "create a sub-slice of `Path` from the 1st character to the end." This drops the leading `/` from the path name.

If you run this program and access the URL:

```
http://localhost:8080/monkeys
```

the program would present a page containing:

Hi there, I love monkeys!

Using net/http to serve wiki pages

To use the net/http package, it must be imported:

```
import (  
    "fmt"  
    "net/http"  
    "io/ioutil"  
)
```

Let's create a handler to view a wiki page:

```
const lenPath = len("/view/")  
  
func viewHandler(w http.ResponseWriter, r *http.Request) {  
    title := r.URL.Path[lenPath:]  
    p, _ := loadPage(title)  
    fmt.Fprintf(w, "<h1>%s</h1><div>%s</div>", p.Title, p.Body)  
}
```

First, this function extracts the page title from `r.URL.Path`, the path component of the request URL. The global constant `lenPath` is the length of the leading `"/view/"` component of the request path. The `Path` is re-sliced with `[lenPath:]` to drop the first 6 characters of the string. This is because the path will invariably begin with `"/view/"`, which is not part of the page title.

The function then loads the page data, formats the page with a string of simple HTML, and writes it to `w`, the `http.ResponseWriter`.

Again, note the use of `_` to ignore the error return value from `loadPage`. This is done here for simplicity and generally considered bad practice. We will attend to this later.

To use this handler, we create a main function that initializes `http` using the `viewHandler` to handle any requests under the path `/view/`.

```
func main() {  
    http.HandleFunc("/view/", viewHandler)  
    http.ListenAndServe(":8080", nil)  
}
```

[Click here to view the code we've written so far.](#)

Let's create some page data (as `test.txt`), compile our code, and try serving a wiki page.

Open `test.txt` file in your editor, and save the string "Hello world" (without quotes) in it.

```
$ go build wiki.go  
$ ./wiki
```

With this web server running, a visit to <http://localhost:8080/view/test> should show a page titled "test" containing the words "Hello world".

Editing Pages

A wiki is not a wiki without the ability to edit pages. Let's create two new handlers: one named `editHandler` to display an 'edit page' form, and the other named `saveHandler` to save the data entered via the form.

First, we add them to `main()`:

```
func main() {
    http.HandleFunc("/view/", viewHandler)
    http.HandleFunc("/edit/", editHandler)
    http.HandleFunc("/save/", saveHandler)
    http.ListenAndServe(":8080", nil)
}
```

The function `editHandler` loads the page (or, if it doesn't exist, create an empty `Page` struct), and displays an HTML form.

```
func editHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    fmt.Fprintf(w, "<h1>Editing %s</h1>"+
        "<form action=\"/save/%s\" method=\"POST\">"+
        "<textarea name=\"body\">%s</textarea><br>"+
        "<input type=\"submit\" value=\"Save\">"+
        "</form>",
        p.Title, p.Title, p.Body)
}
```

This function will work fine, but all that hard-coded HTML is ugly. Of course, there is a better way.

The html/template package

The `html/template` package is part of the Go standard library. We can use `html/template` to keep the HTML in a separate file, allowing us to change the layout of our edit page without modifying the underlying Go code.

First, we must add `html/template` to the list of imports:

```
import (  
    "html/template"  
    "http"  
    "io/ioutil"  
    "os"  
)
```

Let's create a template file containing the HTML form. Open a new file named `edit.html`, and add the following lines:

```
<h1>Editing {{.Title}}</h1>  
  
<form action="/save/{{.Title}}" method="POST">  
<div><textarea name="body" rows="20" cols="80">{{printf "%s" .Body}}  
<div><input type="submit" value="Save"></div>  
</form>
```

Modify `editHandler` to use the template, instead of the hard-coded HTML:

```
func editHandler(w http.ResponseWriter, r *http.Request) {  
    title := r.URL.Path[lenPath:]  
    p, err := loadPage(title)  
    if err != nil {  
        p = &Page{Title: title}  
    }  
    t, _ := template.ParseFiles("edit.html")  
    t.Execute(w, p)  
}
```

The function `template.ParseFiles` will read the contents of `edit.html` and return a `*template.Template`.

The method `t.Execute` executes the template, writing the generated HTML to the `http.ResponseWriter`. The `.Title` and `.Body` dotted identifiers refer to `p.Title` and `p.Body`.

Template directives are enclosed in double curly braces. The `printf "%s" .Body` instruction is a function call that outputs `.Body` as a string instead of a stream of bytes, the same as a call to `fmt.Printf`. The `html/template` package helps guarantee that only safe and correct-looking HTML is generated by template actions. For instance, it automatically escapes any greater than sign (`>`), replacing it with `>`, to make sure user data does not corrupt the form HTML.

Now that we've removed the `fmt.Fprintf` statement, we can remove `"fmt"` from the `import` list.

While we're working with templates, let's create a template for our `viewHandler` called `view.html`:

```
<h1>{{.Title}}</h1>
<p>[<a href="/edit/{{.Title}}">edit</a>]</p>
<div>{{printf "%s" .Body}}</div>
```

Modify `viewHandler` accordingly:

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, _ := loadPage(title)
    t, _ := template.ParseFiles("view.html")
    t.Execute(w, p)
}
```

Notice that we've used almost exactly the same templating code in both handlers. Let's remove this duplication by moving the templating code to its own function:

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, _ := loadPage(title)
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    renderTemplate(w, "edit", p)
}
```

```
func renderTemplate(w http.ResponseWriter, tmpl string, p *Page) {  
    t, _ := template.ParseFiles(tmpl + ".html")  
    t.Execute(w, p)  
}
```

The handlers are now shorter and simpler.

Handling non-existent pages

What if you visit [/view/APageThatDoesntExist](#)? The program will crash. This is because it ignores the error return value from `loadPage`. Instead, if the requested Page doesn't exist, it should redirect the client to the edit Page so the content may be created:

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}
```

The `http.Redirect` function adds an HTTP status code of `http.StatusFound` (302) and a `Location` header to the HTTP response.

Saving Pages

The function `saveHandler` will handle the form submission.

```
func saveHandler(w http.ResponseWriter, r *http.Request) {
    title := r.URL.Path[lenPath:]
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    p.save()
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

The page title (provided in the URL) and the form's only field, `Body`, are stored in a new `Page`. The `save()` method is then called to write the data to a file, and the client is redirected to the `/view/` page.

The value returned by `FormValue` is of type `string`. We must convert that value to `[]byte` before it will fit into the `Page` struct. We use `[]byte(body)` to perform the conversion.

Error handling

There are several places in our program where errors are being ignored. This is bad practice, not least because when an error does occur the program will crash. A better solution is to handle the errors and return an error message to the user. That way if something does go wrong, the server will continue to function and the user will be notified.

First, let's handle the errors in `renderTemplate`:

```
func renderTemplate(w http.ResponseWriter, tmpl string, p *Page) {
    t, err := template.ParseFiles(tmpl + ".html")
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    err = t.Execute(w, p)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

The `http.Error` function sends a specified HTTP response code (in this case "Internal Server Error") and error message. Already the decision to put this in a separate function is paying off.

Now let's fix up `saveHandler`:

```
func saveHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    err = p.save()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

Any errors that occur during `p.save()` will be reported to the user.

Template caching

There is an inefficiency in this code: `renderTemplate` calls `ParseFiles` every time a page is rendered. A better approach would be to call `ParseFiles` once at program initialization, parsing all templates into a single `*Template`. Then we can use the [ExecuteTemplate](#) method to render a specific template.

First we create a global variable named `templates`, and initialize it with `ParseFiles`.

```
var templates = template.Must(template.ParseFiles("edit.html", "view
```

The function `template.Must` is a convenience wrapper that panics when passed a non-nil error value, and otherwise returns the `*Template` unaltered. A panic is appropriate here; if the templates can't be loaded the only sensible thing to do is exit the program.

A for loop is used with a range statement to iterate over an array constant containing the names of the templates we want parsed. If we were to add more templates to our program, we would add their names to that array.

We then modify the `renderTemplate` function to call the `templates.ExecuteTemplate` method with the name of the appropriate template:

```
func renderTemplate(w http.ResponseWriter, tmpl string, p *Page) {
    err := templates.ExecuteTemplate(w, tmpl+".html", p)
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

Note that the template name is the template file name, so we must append `".html"` to the `tmpl` argument.

Validation

As you may have observed, this program has a serious security flaw: a user can supply an arbitrary path to be read/written on the server. To mitigate this, we can write a function to validate the title with a regular expression.

First, add "regexp" to the import list. Then we can create a global variable to store our validation regexp:

```
var titleValidator = regexp.MustCompile("^[a-zA-Z0-9]+$")
```

The function `regexp.MustCompile` will parse and compile the regular expression, and return a `regexp.Regexp`. `MustCompile` is distinct from `Compile` in that it will panic if the expression compilation fails, while `Compile` returns an error as a second parameter.

Now, let's write a function that extracts the title string from the request URL, and tests it against our `TitleValidator` expression:

```
func getTitle(w http.ResponseWriter, r *http.Request) (title string,
    title = r.URL.Path[lenPath:]
    if !titleValidator.MatchString(title) {
        http.NotFound(w, r)
        err = errors.New("Invalid Page Title")
    }
    return
}
```

If the title is valid, it will be returned along with a `nil` error value. If the title is invalid, the function will write a "404 Not Found" error to the HTTP connection, and return an error to the handler.

Let's put a call to `getTitle` in each of the handlers:

```
func viewHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
    }
}
```

```

        return
    }
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    renderTemplate(w, "edit", p)
}

func saveHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    err = p.save()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}

```

Introducing Function Literals and Closures

Catching the error condition in each handler introduces a lot of repeated code. What if we could wrap each of the handlers in a function that does this validation and error checking? Go's [function literals](#) provide a powerful means of abstracting functionality that can help us here.

First, we re-write the function definition of each of the handlers to accept a title string:

```
func viewHandler(w http.ResponseWriter, r *http.Request, title string) {
func editHandler(w http.ResponseWriter, r *http.Request, title string) {
func saveHandler(w http.ResponseWriter, r *http.Request, title string) {
```

Now let's define a wrapper function that *takes a function of the above type*, and returns a function of type `http.HandlerFunc` (suitable to be passed to the function `http.HandleFunc`):

```
func makeHandler(fn func(http.ResponseWriter, *http.Request, string)
    return func(w http.ResponseWriter, r *http.Request) {
        // Here we will extract the page title from the Request
        // and call the provided handler 'fn'
    }
}
```

The returned function is called a closure because it encloses values defined outside of it. In this case, the variable `fn` (the single argument to `makeHandler`) is enclosed by the closure. The variable `fn` will be one of our `save`, `edit`, or `view` handlers.

Now we can take the code from `getTitle` and use it here (with some minor modifications):

```
func makeHandler(fn func(http.ResponseWriter, *http.Request, string)
    return func(w http.ResponseWriter, r *http.Request) {
        title := r.URL.Path[lenPath:]
        if !titleValidator.MatchString(title) {
            http.NotFound(w, r)
            return
        }
        fn(w, r, title)
    }
}
```

```
}
```

The closure returned by `makeHandler` is a function that takes an `http.ResponseWriter` and `http.Request` (in other words, an `http.HandlerFunc`). The closure extracts the `title` from the request path, and validates it with the `titleValidator` regexp. If the `title` is invalid, an error will be written to the `ResponseWriter` using the `http.NotFound` function. If the `title` is valid, the enclosed handler function `fn` will be called with the `ResponseWriter`, `Request`, and `title` as arguments.

Now we can wrap the handler functions with `makeHandler` in `main`, before they are registered with the `http` package:

```
func main() {
    http.HandleFunc("/view/", makeHandler(viewHandler))
    http.HandleFunc("/edit/", makeHandler(editHandler))
    http.HandleFunc("/save/", makeHandler(saveHandler))
    http.ListenAndServe(":8080", nil)
}
```

Finally we remove the calls to `getTitle` from the handler functions, making them much simpler:

```
func viewHandler(w http.ResponseWriter, r *http.Request, title string,
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}
```

```
func editHandler(w http.ResponseWriter, r *http.Request, title string,
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    renderTemplate(w, "edit", p)
}
```

```
func saveHandler(w http.ResponseWriter, r *http.Request, title string,
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    err := p.save()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}
```

```
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

Try it out!

[Click here to view the final code listing.](#)

Recompile the code, and run the app:

```
$ go build wiki.go  
$ ./wiki
```

Visiting <http://localhost:8080/view/ANewPage> should present you with the page edit form. You should then be able to enter some text, click 'Save', and be redirected to the newly created page.

Other tasks

Here are some simple tasks you might want to tackle on your own:

- Store templates in `tmpl/` and page data in `data/`.
- Add a handler to make the web root redirect to `/view/FrontPage`.
- Spruce up the page templates by making them valid HTML and adding some CSS rules.
- Implement inter-page linking by converting instances of `[PageName]` to `PageName`. (hint: you could use `regexp.ReplaceAllFunc` to do this)

Build version `go1.0.1`.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

JSON-RPC: a tale of interfaces

Here we present an example where Go's [interfaces](#) made it easy to refactor some existing code to make it more flexible and extensible. Originally, the standard library's [RPC package](#) used a custom wire format called [gob](#). For a particular application, we wanted to use [JSON](#) as an alternate wire format.

We first defined a pair of interfaces to describe the functionality of the existing wire format, one for the client, and one for the server (depicted below).

```
type ServerCodec interface {
    ReadRequestHeader(*Request) error
    ReadRequestBody(interface{}) error
    WriteResponse(*Response, interface{}) error
    Close() error
}
```

On the server side, we then changed two internal function signatures to accept the `ServerCodec` interface instead of our existing `gob.Encoder`. Here's one of them:

```
func sendResponse(sending *sync.Mutex, req *Request,
    reply interface{}, enc *gob.Encoder, errmsg string)
```

became

```
func sendResponse(sending *sync.Mutex, req *Request,
    reply interface{}, enc ServerCodec, errmsg string)
```

We then wrote a trivial `gobServerCodec` wrapper to reproduce the original functionality. From there it is simple to build a `jsonServerCodec`.

After some similar changes to the client side, this was the full extent of the work we needed to do on the `RPC` package. This whole exercise took about 20 minutes! After tidying up and testing the new code, the [final changeset](#) was submitted.

In an inheritance-oriented language like Java or C++, the obvious path would be to generalize the `RPC` class, and create `JsonRPC` and `GobRPC` subclasses. However, this approach becomes tricky if you want to make a further

generalization orthogonal to that hierarchy. (For example, if you were to implement an alternate RPC standard). In our Go package, we took a route that is both conceptually simpler and requires less code be written or changed.

A vital quality for any codebase is maintainability. As needs change, it is essential to adapt your code easily and cleanly, lest it become unwieldy to work with. We believe Go's lightweight, composition-oriented type system provides a means of structuring code that scales.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Go's Declaration Syntax

Newcomers to Go wonder why the declaration syntax is different from the tradition established in the C family. In this post we'll compare the two approaches and explain why Go's declarations look as they do.

C syntax

First, let's talk about C syntax. C took an unusual and clever approach to declaration syntax. Instead of describing the types with special syntax, one writes an expression involving the item being declared, and states what type that expression will have. Thus

```
int x;
```

declares `x` to be an `int`: the expression '`x`' will have type `int`. In general, to figure out how to write the type of a new variable, write an expression involving that variable that evaluates to a basic type, then put the basic type on the left and the expression on the right.

Thus, the declarations

```
int *p;  
int a[3];
```

state that `p` is a pointer to `int` because '`*p`' has type `int`, and that `a` is an array of `ints` because `a[3]` (ignoring the particular index value, which is punned to be the size of the array) has type `int`.

What about functions? Originally, C's function declarations wrote the types of the arguments outside the parens, like this:

```
int main(argc, argv)  
    int argc;  
    char *argv[];  
{ /* ... */ }
```

Again, we see that `main` is a function because the expression `main(argc, argv)` returns an `int`. In modern notation we'd write

```
int main(int argc, char *argv[]) { /* ... */ }
```

but the basic structure is the same.

This is a clever syntactic idea that works well for simple types but can get confusing fast. The famous example is declaring a function pointer. Follow the rules and you get this:

```
int (*fp)(int a, int b);
```

Here, fp is a pointer to a function because if you write the expression (*fp)(a, b) you'll call a function that returns int. What if one of fp's arguments is itself a function?

```
int (*fp)(int (*ff)(int x, int y), int b)
```

That's starting to get hard to read.

Of course, we can leave out the name of the parameters when we declare a function, so main can be declared

```
int main(int, char *[])
```

Recall that argv is declared like this,

```
char *argv[]
```

so you drop the name from the *middle* of its declaration to construct its type. It's not obvious, though, that you declare something of type char *[] by putting its name in the middle.

And look what happens to fp's declaration if you don't name the parameters:

```
int (*fp)(int (*)(int, int), int)
```

Not only is it not obvious where to put the name inside

```
int (*)(int, int)
```

it's not exactly clear that it's a function pointer declaration at all. And what if the return type is a function pointer?

```
int ((*fp)(int (*)(int, int), int))(int, int)
```

It's hard even to see that this declaration is about fp.

You can construct more elaborate examples but these should illustrate some of the difficulties that C's declaration syntax can introduce.

There's one more point that needs to be made, though. Because type and declaration syntax are the same, it can be difficult to parse expressions with types in the middle. This is why, for instance, C casts always parenthesize the type, as in

```
(int)M_PI
```

Go syntax

Languages outside the C family usually use a distinct type syntax in declarations. Although it's a separate point, the name usually comes first, often followed by a colon. Thus our examples above become something like (in a fictional but illustrative language)

```
x: int  
p: pointer to int  
a: array[3] of int
```

These declarations are clear, if verbose - you just read them left to right. Go takes its cue from here, but in the interests of brevity it drops the colon and removes some of the keywords:

```
x int  
p *int  
a [3]int
```

There is no direct correspondence between the look of [3]int and how to use a in an expression. (We'll come back to pointers in the next section.) You gain clarity at the cost of a separate syntax.

Now consider functions. Let's transcribe the declaration for main, even though the main function in Go takes no arguments:

```
func main(argc int, argv *[]byte) int
```

Superficially that's not much different from C, but it reads well from left to right:

function main takes an int and a pointer to a slice of bytes and returns an int.

Drop the parameter names and it's just as clear - they're always first so there's no confusion.

```
func main(int, *[]byte) int
```

One value of this left-to-right style is how well it works as the types become more complex. Here's a declaration of a function variable (analogous to a function pointer in C):

```
f func(func(int,int) int, int) int
```

Or if f returns a function:

```
f func(func(int,int) int, int) func(int, int) int
```

It still reads clearly, from left to right, and it's always obvious which name is being declared - the name comes first.

The distinction between type and expression syntax makes it easy to write and invoke closures in Go:

```
sum := func(a, b int) int { return a+b } (3, 4)
```

Pointers

Pointers are the exception that proves the rule. Notice that in arrays and slices, for instance, Go's type syntax puts the brackets on the left of the type but the expression syntax puts them on the right of the expression:

```
var a []int  
x = a[1]
```

For familiarity, Go's pointers use the * notation from C, but we could not bring ourselves to make a similar reversal for pointer types. Thus pointers work like this

```
var p *int  
x = *p
```

We couldn't say

```
var p *int
x = p*
```

because that postfix `*` would conflate with multiplication. We could have used the Pascal `^`, for example:

```
var p ^int
x = p^
```

and perhaps we should have (and chosen another operator for xor), because the prefix asterisk on both types and expressions complicates things in a number of ways. For instance, although one can write

```
[]int("hi")
```

as a conversion, one must parenthesize the type if it starts with a `*`:

```
(*int)(nil)
```

Had we been willing to give up `*` as pointer syntax, those parentheses would be unnecessary.

So Go's pointer syntax is tied to the familiar C form, but those ties mean that we cannot break completely from using parentheses to disambiguate types and expressions in the grammar.

Overall, though, we believe Go's type syntax is easier to understand than C's, especially when things get complicated.

Notes

Go's declarations read left to right. It's been pointed out that C's read in a spiral! See [The "Clockwise/Spiral Rule"](#) by David Anderson.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Defer, Panic, and Recover

Go has the usual mechanisms for control flow: if, for, switch, goto. It also has the go statement to run code in a separate goroutine. Here I'd like to discuss some of the less common ones: defer, panic, and recover.

A **defer statement** pushes a function call onto a list. The list of saved calls is executed after the surrounding function returns. Defer is commonly used to simplify functions that perform various clean-up actions.

For example, let's look at a function that opens two files and copies the contents of one file to the other:

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }

    written, err = io.Copy(dst, src)
    dst.Close()
    src.Close()
    return
}
```

This works, but there is a bug. If the call to `os.Create` fails, the function will return without closing the source file. This can be easily remedied by putting a call to `src.Close` before the second return statement, but if the function were more complex the problem might not be so easily noticed and resolved. By introducing defer statements we can ensure that the files are always closed:

```
func CopyFile(dstName, srcName string) (written int64, err error) {
    src, err := os.Open(srcName)
    if err != nil {
        return
    }
    defer src.Close()

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }

    written, err = io.Copy(dst, src)
    dst.Close()
    return
}
```

```

    dst, err := os.Create(dstName)
    if err != nil {
        return
    }
    defer dst.Close()

    return io.Copy(dst, src)
}

```

Defer statements allow us to think about closing each file right after opening it, guaranteeing that, regardless of the number of return statements in the function, the files *will* be closed.

The behavior of defer statements is straightforward and predictable. There are three simple rules:

1. *A deferred function's arguments are evaluated when the defer statement is evaluated.*

In this example, the expression "i" is evaluated when the Println call is deferred. The deferred call will print "0" after the function returns.

```

func a() {
    i := 0
    defer fmt.Println(i)
    i++
    return
}

```

2. *Deferred function calls are executed in Last In First Out order after the surrounding function returns.*

This function prints "3210":

```

func b() {
    for i := 0; i < 4; i++ {
        defer fmt.Print(i)
    }
}

```

3. *Deferred functions may read and assign to the returning function's named return values.*

In this example, a deferred function increments the return value *i* *after* the surrounding function returns. Thus, this function returns 2:

```
func c() (i int) {
    defer func() { i++ }()
    return 1
}
```

This is convenient for modifying the error return value of a function; we will see an example of this shortly.

Panic is a built-in function that stops the ordinary flow of control and begins *panicking*. When the function *F* calls `panic`, execution of *F* stops, any deferred functions in *F* are executed normally, and then *F* returns to its caller. To the caller, *F* then behaves like a call to `panic`. The process continues up the stack until all functions in the current goroutine have returned, at which point the program crashes. Panics can be initiated by invoking `panic` directly. They can also be caused by runtime errors, such as out-of-bounds array accesses.

Recover is a built-in function that regains control of a panicking goroutine. `Recover` is only useful inside deferred functions. During normal execution, a call to `recover` will return `nil` and have no other effect. If the current goroutine is panicking, a call to `recover` will capture the value given to `panic` and resume normal execution.

Here's an example program that demonstrates the mechanics of `panic` and `defer`:

```
package main

import "fmt"

func main() {
    f()
    fmt.Println("Returned normally from f.")
}

func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    fmt.Println("Calling g.")
    g(0)
}
```

```

    fmt.Println("Returned normally from g.")
}

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v", i))
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}

```

The function `g` takes the `int` `i`, and panics if `i` is greater than 3, or else it calls itself with the argument `i+1`. The function `f` defers a function that calls `recover` and prints the recovered value (if it is non-nil). Try to picture what the output of this program might be before reading on.

The program will output:

```

Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
Recovered in f 4
Returned normally from f.

```

If we remove the deferred function from `f` the panic is not recovered and reaches the top of the goroutine's call stack, terminating the program. This modified program will output:

```

Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0

```

```
panic: 4
```

```
panic PC=0x2a9cd8  
[stack trace omitted]
```

For a real-world example of **panic** and **recover**, see the [json package](#) from the Go standard library. It decodes JSON-encoded data with a set of recursive functions. When malformed JSON is encountered, the parser calls panic to unwind the stack to the top-level function call, which recovers from the panic and returns an appropriate error value (see the 'error' and 'unmarshal' methods of the decodeState type in [decode.go](#)).

The convention in the Go libraries is that even when a package uses panic internally, its external API still presents explicit error return values.

Other uses of **defer** (beyond the file.Close example given earlier) include releasing a mutex:

```
mu.Lock()  
defer mu.Unlock()
```

printing a footer:

```
printHeader()  
defer printFooter()
```

and more.

In summary, the defer statement (with or without panic and recover) provides an unusual and powerful mechanism for control flow. It can be used to model a number of features implemented by special-purpose structures in other programming languages. Try it out.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Go Concurrency Patterns: Timing out, moving on

Concurrent programming has its own idioms. A good example is timeouts. Although Go's channels do not support them directly, they are easy to implement. Say we want to receive from the channel `ch`, but want to wait at most one second for the value to arrive. We would start by creating a signalling channel and launching a goroutine that sleeps before sending on the channel:

```
timeout := make(chan bool, 1)
go func() {
    time.Sleep(1 * time.Second)
    timeout <- true
}()
```

We can then use a `select` statement to receive from either `ch` or `timeout`. If nothing arrives on `ch` after one second, the `timeout` case is selected and the attempt to read from `ch` is abandoned.

```
select {
case <-ch:
    // a read from ch has occurred
case <-timeout:
    // the read from ch has timed out
}
```

The `timeout` channel is buffered with space for 1 value, allowing the `timeout` goroutine to send to the channel and then exit. The goroutine doesn't know (or care) whether the value is received. This means the goroutine won't hang around forever if the `ch` receive happens before the timeout is reached. The `timeout` channel will eventually be deallocated by the garbage collector.

(In this example we used `time.Sleep` to demonstrate the mechanics of goroutines and channels. In real programs you should use [time.After](#), a function that returns a channel and sends on that channel after the specified duration.)

Let's look at another variation of this pattern. In this example we have a program

that reads from multiple replicated databases simultaneously. The program needs only one of the answers, and it should accept the answer that arrives first.

The function `Query` takes a slice of database connections and a query string. It queries each of the databases in parallel and returns the first response it receives:

```
func Query(conns []Conn, query string) Result {
    ch := make(chan Result, 1)
    for _, conn := range conns {
        go func(c Conn) {
            select {
                case ch <- c.DoQuery(query):
                default:
            }
        }(conn)
    }
    return <-ch
}
```

In this example, the closure does a non-blocking send, which it achieves by using the `send` operation in `select` statement with a `default` case. If the send cannot go through immediately the default case will be selected. Making the send non-blocking guarantees that none of the goroutines launched in the loop will hang around. However, if the result arrives before the main function has made it to the receive, the send could fail since no one is ready.

This problem is a textbook example of what is known as a [race condition](#), but the fix is trivial. We just make sure to buffer the channel `ch` (by adding the buffer length as the second argument to [make](#)), guaranteeing that the first send has a place to put the value. This ensures the send will always succeed, and the first value to arrive will be retrieved regardless of the order of execution.

These two examples demonstrate the simplicity with which Go can express complex interactions between goroutines.

Build version `go1.0.1`.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Slices: usage and internals

Go's slice type provides a convenient and efficient means of working with sequences of typed data. Slices are analogous to arrays in other languages, but have some unusual properties. This article will look at what slices are and how they are used.

Arrays

The slice type is an abstraction built on top of Go's array type, and so to understand slices we must first understand arrays.

An array type definition specifies a length and an element type. For example, the type `[4]int` represents an array of four integers. An array's size is fixed; its length is part of its type (`[4]int` and `[5]int` are distinct, incompatible types). Arrays can be indexed in the usual way, so the expression `s[n]` accesses the *n*th element:

```
var a [4]int
a[0] = 1
i := a[0]
// i == 1
```

Arrays do not need to be initialized explicitly; the zero value of an array is a ready-to-use array whose elements are themselves zeroed:

```
// a[2] == 0, the zero value of the int type
```

The in-memory representation of `[4]int` is just four integer values laid out sequentially:



Go's arrays are values. An array variable denotes the entire array; it is not a pointer to the first array element (as would be the case in C). This means that when you assign or pass around an array value you will make a copy of its contents. (To avoid the copy you could pass a *pointer* to the array, but then that's a pointer to an array, not an array.) One way to think about arrays is as a sort of struct but with indexed rather than named fields: a fixed-size composite value.

An array literal can be specified like so:

```
b := [2]string{"Penn", "Teller"}
```

Or, you can have the compiler count the array elements for you:

```
b := [...]string{"Penn", "Teller"}
```

In both cases, the type of `b` is `[2]string`.

Slices

Arrays have their place, but they're a bit inflexible, so you don't see them too often in Go code. Slices, though, are everywhere. They build on arrays to provide great power and convenience.

The type specification for a slice is `[]T`, where `T` is the type of the elements of the slice. Unlike an array type, a slice type has no specified length.

A slice literal is declared just like an array literal, except you leave out the element count:

```
letters := []string{"a", "b", "c", "d"}
```

A slice can be created with the built-in function called `make`, which has the signature,

```
func make([]T, len, cap) []T
```

where `T` stands for the element type of the slice to be created. The `make` function takes a type, a length, and an optional capacity. When called, `make` allocates an array and returns a slice that refers to that array.

```
var s []byte
s = make([]byte, 5, 5)
// s == []byte{0, 0, 0, 0, 0}
```

When the capacity argument is omitted, it defaults to the specified length. Here's a more succinct version of the same code:

```
s := make([]byte, 5)
```

The length and capacity of a slice can be inspected using the built-in `len` and `cap` functions.

```
len(s) == 5
cap(s) == 5
```

The next two sections discuss the relationship between length and capacity.

The zero value of a slice is `nil`. The `len` and `cap` functions will both return 0 for a `nil` slice.

A slice can also be formed by "slicing" an existing slice or array. Slicing is done by specifying a half-open range with two indices separated by a colon. For example, the expression `b[1:4]` creates a slice including elements 1 through 3 of `b` (the indices of the resulting slice will be 0 through 2).

```
b := []byte{'g', 'o', 'l', 'a', 'n', 'g'}
// b[1:4] == []byte{'o', 'l', 'a'}, sharing the same storage as b
```

The start and end indices of a slice expression are optional; they default to zero and the slice's length respectively:

```
// b[:2] == []byte{'g', 'o'}
// b[2:] == []byte{'l', 'a', 'n', 'g'}
// b[:] == b
```

This is also the syntax to create a slice given an array:

```
x := [3]string{"000A", "000", "00"}
s := x[:] // a slice referencing the storage of x
```

Slice internals

A slice is a descriptor of an array segment. It consists of a pointer to the array, the length of the segment, and its capacity (the maximum length of the segment).



Our variable `s`, created earlier by `make([]byte, 5)`, is structured like this:



The length is the number of elements referred to by the slice. The capacity is the

number of elements in the underlying array (beginning at the element referred to by the slice pointer). The distinction between length and capacity will be made clear as we walk through the next few examples.

As we slice `s`, observe the changes in the slice data structure and their relation to the underlying array:

```
s = s[2:4]
```



Slicing does not copy the slice's data. It creates a new slice value that points to the original array. This makes slice operations as efficient as manipulating array indices. Therefore, modifying the *elements* (not the slice itself) of a re-slice modifies the elements of the original slice:

```
d := []byte{'r', 'o', 'a', 'd'}
e := d[2:]
// e == []byte{'a', 'd'}
e[1] == 'm'
// e == []byte{'a', 'm'}
// d == []byte{'r', 'o', 'a', 'm'}
```

Earlier we sliced `s` to a length shorter than its capacity. We can grow `s` to its capacity by slicing it again:

```
s = s[:cap(s)]
```



A slice cannot be grown beyond its capacity. Attempting to do so will cause a runtime panic, just as when indexing outside the bounds of a slice or array. Similarly, slices cannot be re-sliced below zero to access earlier elements in the array.

Growing slices (the copy and append functions)

To increase the capacity of a slice one must create a new, larger slice and copy the contents of the original slice into it. This technique is how dynamic array implementations from other languages work behind the scenes. The next example doubles the capacity of `s` by making a new slice, `t`, copying the contents of `s` into `t`, and then assigning the slice value `t` to `s`:

```
t := make([]byte, len(s), (cap(s)+1)*2) // +1 in case cap(s) == 0
for i := range s {
    t[i] = s[i]
}
s = t
```

The looping piece of this common operation is made easier by the built-in copy function. As the name suggests, copy copies data from a source slice to a destination slice. It returns the number of elements copied.

```
func copy(dst, src []T) int
```

The copy function supports copying between slices of different lengths (it will copy only up to the smaller number of elements). In addition, copy can handle source and destination slices that share the same underlying array, handling overlapping slices correctly.

Using copy, we can simplify the code snippet above:

```
t := make([]byte, len(s), (cap(s)+1)*2)
copy(t, s)
s = t
```

A common operation is to append data to the end of a slice. This function appends byte elements to a slice of bytes, growing the slice if necessary, and returns the updated slice value:

```
func AppendByte(slice []byte, data ...byte) []byte {
    m := len(slice)
    n := m + len(data)
    if n > cap(slice) { // if necessary, reallocate
        // allocate double what's needed, for future growth.
        newSlice := make([]byte, (n+1)*2)
        copy(newSlice, slice)
        slice = newSlice
    }
    slice = slice[0:n]
    copy(slice[m:n], data)
    return slice
}
```

One could use AppendByte like this:

```
p := []byte{2, 3, 5}
p = AppendByte(p, 7, 11, 13)
```

```
// p == []byte{2, 3, 5, 7, 11, 13}
```

Functions like `AppendByte` are useful because they offer complete control over the way the slice is grown. Depending on the characteristics of the program, it may be desirable to allocate in smaller or larger chunks, or to put a ceiling on the size of a reallocation.

But most programs don't need complete control, so Go provides a built-in `append` function that's good for most purposes; it has the signature

```
func append(s []T, x ...T) []T
```

The `append` function appends the elements `x` to the end of the slice `s`, and grows the slice if a greater capacity is needed.

```
a := make([]int, 1)
// a == []int{0}
a = append(a, 1, 2, 3)
// a == []int{0, 1, 2, 3}
```

To append one slice to another, use `...` to expand the second argument to a list of arguments.

```
a := []string{"John", "Paul"}
b := []string{"George", "Ringo", "Pete"}
a = append(a, b...) // equivalent to "append(a, b[0], b[1], b[2])"
// a == []string{"John", "Paul", "George", "Ringo", "Pete"}
```

Since the zero value of a slice (`nil`) acts like a zero-length slice, you can declare a slice variable and then append to it in a loop:

```
// Filter returns a new slice holding only
// the elements of s that satisfy f()
func Filter(s []int, fn func(int) bool) []int {
    var p []int // == nil
    for _, i := range s {
        if fn(i) {
            p = append(p, i)
        }
    }
    return p
}
```

A possible "gotcha"

As mentioned earlier, re-slicing a slice doesn't make a copy of the underlying array. The full array will be kept in memory until it is no longer referenced. Occasionally this can cause the program to hold all the data in memory when only a small piece of it is needed.

For example, this `FindDigits` function loads a file into memory and searches it for the first group of consecutive numeric digits, returning them as a new slice.

```
var digitRegexp = regexp.MustCompile("[0-9]+")

func FindDigits(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    return digitRegexp.Find(b)
}
```

This code behaves as advertised, but the returned `[]byte` points into an array containing the entire file. Since the slice references the original array, as long as the slice is kept around the garbage collector can't release the array; the few useful bytes of the file keep the entire contents in memory.

To fix this problem one can copy the interesting data to a new slice before returning it:

```
func CopyDigits(filename string) []byte {
    b, _ := ioutil.ReadFile(filename)
    b = digitRegexp.Find(b)
    c := make([]byte, len(b))
    copy(c, b)
    return c
}
```

A more concise version of this function could be constructed by using `append`. This is left as an exercise for the reader.

Further Reading

[Effective Go](#) contains an in-depth treatment of [slices](#) and [arrays](#), and the [Go language specification](#) defines [slices](#) and their [associated helper functions](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Error Handling and Go

If you have written any Go code you have probably encountered the built-in error type. Go code uses error values to indicate an abnormal state. For example, the `os.Open` function returns a non-nil error value when it fails to open a file.

```
func Open(name string) (file *File, err error)
```

The following code uses `os.Open` to open a file. If an error occurs it calls `log.Fatal` to print the error message and stop.

```
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
// do something with the open *File f
```

You can get a lot done in Go knowing just this about the error type, but in this article we'll take a closer look at error and discuss some good practices for error handling in Go.

The error type

The error type is an interface type. An error variable represents any value that can describe itself as a string. Here is the interface's declaration:

```
type error interface {
    Error() string
}
```

The error type, as with all built in types, is [predeclared](#) in the [universe block](#).

The most commonly-used error implementation is the [errors](#) package's unexported `errorString` type.

```
// errorString is a trivial implementation of error.
type errorString struct {
    s string
}
```

```
func (e *errorString) Error() string {
    return e.s
}
```

You can construct one of these values with the `errors.New` function. It takes a string that it converts to an `errors.errorString` and returns as an error value.

```
// New returns an error that formats as the given text.
func New(text string) error {
    return &errorString{text}
}
```

Here's how you might use `errors.New`:

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number")
    }
    // implementation
}
```

A caller passing a negative argument to `Sqrt` receives a non-nil error value (whose concrete representation is an `errors.errorString` value). The caller can access the error string ("math: square root of...") by calling the error's `Error` method, or by just printing it:

```
f, err := Sqrt(-1)
if err != nil {
    fmt.Println(err)
}
```

The [fmt](#) package formats an error value by calling its `Error()` string method.

It is the error implementation's responsibility to summarize the context. The error returned by `os.Open` formats as "open /etc/passwd: permission denied," not just "permission denied." The error returned by our `Sqrt` is missing information about the invalid argument.

To add that information, a useful function is the `fmt` package's `Errorf`. It formats a string according to `Printf`'s rules and returns it as an error created by `errors.New`.

```
if f < 0 {
    return 0, fmt.Errorf("math: square root of negative number %
```

```
}
```

In many cases `fmt.Errorf` is good enough, but since `error` is an interface, you can use arbitrary data structures as error values, to allow callers to inspect the details of the error.

For instance, our hypothetical callers might want to recover the invalid argument passed to `Sqrt`. We can enable that by defining a new error implementation instead of using `errors.Errorf`:

```
type NegativeSqrtError float64

func (f NegativeSqrtError) Error() string {
    return fmt.Sprintf("math: square root of negative number %g", f)
}
```

A sophisticated caller can then use a [type assertion](#) to check for a `NegativeSqrtError` and handle it specially, while callers that just pass the error to `fmt.Println` or `log.Fatal` will see no change in behavior.

As another example, the [json](#) package specifies a `SyntaxError` type that the `json.Decode` function returns when it encounters a syntax error parsing a JSON blob.

```
type SyntaxError struct {
    msg      string // description of error
    Offset int64  // error occurred after reading Offset bytes
}

func (e *SyntaxError) Error() string { return e.msg }
```

The `Offset` field isn't even shown in the default formatting of the error, but callers can use it to add file and line information to their error messages:

```
if err := dec.Decode(&val); err != nil {
    if serr, ok := err.(*json.SyntaxError); ok {
        line, col := findLine(f, serr.Offset)
        return fmt.Errorf("%s:%d:%d: %v", f.Name(), line, col, e)
    }
    return err
}
```

(This is a slightly simplified version of some [actual code](#) from the [Camlistore](#) project.)

The error interface requires only a `Error` method; specific error implementations might have additional methods. For instance, the [net](#) package returns errors of type `error`, following the usual convention, but some of the error implementations have additional methods defined by the `net.Error` interface:

```
package net

type Error interface {
    error
    Timeout() bool // Is the error a timeout?
    Temporary() bool // Is the error temporary?
}
```

Client code can test for a `net.Error` with a type assertion and then distinguish transient network errors from permanent ones. For instance, a web crawler might sleep and retry when it encounters a temporary error and give up otherwise.

```
    if nerr, ok := err.(net.Error); ok && nerr.Temporary() {
        time.Sleep(1e9)
        continue
    }
    if err != nil {
        log.Fatal(err)
    }
```

Simplifying repetitive error handling

In Go, error handling is important. The language's design and conventions encourage you to explicitly check for errors where they occur (as distinct from the convention in other languages of throwing exceptions and sometimes catching them). In some cases this makes Go code verbose, but fortunately there are some techniques you can use to minimize repetitive error handling.

Consider an [App Engine](#) application with an HTTP handler that retrieves a record from the datastore and formats it with a template.

```
func init() {
    http.HandleFunc("/view", viewRecord)
}

func viewRecord(w http.ResponseWriter, r *http.Request) {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
```

```

    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        http.Error(w, err.Error(), 500)
        return
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        http.Error(w, err.Error(), 500)
    }
}

```

This function handles errors returned by the `datastore.Get` function and `viewTemplate's Execute` method. In both cases, it presents a simple error message to the user with the HTTP status code 500 ("Internal Server Error"). This looks like a manageable amount of code, but add some more HTTP handlers and you quickly end up with many copies of identical error handling code.

To reduce the repetition we can define our own HTTP `appHandler` type that includes an error return value:

```
type appHandler func(http.ResponseWriter, *http.Request) error
```

Then we can change our `viewRecord` function to return errors:

```

func viewRecord(w http.ResponseWriter, r *http.Request) error {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return err
    }
    return viewTemplate.Execute(w, record)
}

```

This is simpler than the original version, but the [http](#) package doesn't understand functions that return error. To fix this we can implement the `http.Handler` interface's `ServeHTTP` method on `appHandler`:

```

func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Reques
    if err := fn(w, r); err != nil {
        http.Error(w, err.Error(), 500)
    }
}

```

The `ServeHTTP` method calls the `appHandler` function and displays the returned

error (if any) to the user. Notice that the method's receiver, `fn`, is a function. (Go can do that!) The method invokes the function by calling the receiver in the expression `fn(w, r)`.

Now when registering `viewRecord` with the `http` package we use the `Handle` function (instead of `HandleFunc`) as `appHandler` is an `http.Handler` (not an `http.HandlerFunc`).

```
func init() {
    http.Handle("/view", appHandler(viewRecord))
}
```

With this basic error handling infrastructure in place, we can make it more user friendly. Rather than just displaying the error string, it would be better to give the user a simple error message with an appropriate HTTP status code, while logging the full error to the App Engine developer console for debugging purposes.

To do this we create an `appError` struct containing an error and some other fields:

```
type appError struct {
    Error    error
    Message  string
    Code     int
}
```

Next we modify the `appHandler` type to return `*appError` values:

```
type appHandler func(http.ResponseWriter, *http.Request) *appError
```

(It's usually a mistake to pass back the concrete type of an error rather than `error`, for reasons discussed in [the Go FAQ](#), but it's the right thing to do here because `ServeHTTP` is the only place that sees the value and uses its contents.)

And make `appHandler`'s `ServeHTTP` method display the `appError`'s `Message` to the user with the correct HTTP status code and log the full `Error` to the developer console:

```
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if e := fn(w, r); e != nil { // e is *appError, not os.Error.
        c := appengine.NewContext(r)
        c.Errorf("%v", e.Error)
    }
}
```

```
        http.Error(w, e.Message, e.Code)
    }
}
```

Finally, we update `viewRecord` to the new function signature and have it return more context when it encounters an error:

```
func viewRecord(w http.ResponseWriter, r *http.Request) *appError {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return &appError{err, "Record not found", 404}
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        return &appError{err, "Can't display record", 500}
    }
    return nil
}
```

This version of `viewRecord` is the same length as the original, but now each of those lines has specific meaning and we are providing a friendlier user experience.

It doesn't end there; we can further improve the error handling in our application. Some ideas:

- give the error handler a pretty HTML template,
- make debugging easier by writing the stack trace to the HTTP response when the user is an administrator,
- write a constructor function for `appError` that stores the stack trace for easier debugging,
- recover from panics inside the `appHandler`, logging the error to the console as "Critical," while telling the user "a serious error has occurred." This is a nice touch to avoid exposing the user to inscrutable error messages caused by programming errors. See the [Defer, Panic, and Recover](#) article for more details.

Conclusion

Proper error handling is an essential requirement of good software. By employing the techniques described in this post you should be able to write more reliable and succinct Go code.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg

Name	Synopsis
archive	
tar	Package tar implements access to tar archives.
zip	Package zip provides support for reading and writing ZIP archives.
bufio	Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.
builtin	Package builtin provides documentation for Go's predeclared identifiers.
bytes	Package bytes implements functions for the manipulation of byte slices.
compress	
bzip2	Package bzip2 implements bzip2 decompression.
flate	Package flate implements the DEFLATE compressed data format, described in RFC 1951.
gzip	Package gzip implements reading and writing of gzip format compressed files, as specified in RFC 1952.
lzw	Package lzw implements the Lempel-Ziv-Welch compressed data format, described in T. A. Welch, "A Technique for High-Performance Data Compression", Computer, 17(6) (June 1984), pp 8-19.
zlib	Package zlib implements reading and writing of zlib format compressed data, as specified in RFC 1950.
container	
heap	Package heap provides heap operations for any type that implements heap.Interface.
list	Package list implements a doubly linked list.
ring	Package ring implements operations on circular lists.
crypto	Package crypto collects common cryptographic constants.

<u>aes</u>	Package aes implements AES encryption (formerly Rijndael), as defined in U.S. Federal Information Processing Standards Publication 197.
<u>cipher</u>	Package cipher implements standard block cipher modes that can be wrapped around low-level block cipher implementations.
<u>des</u>	Package des implements the Data Encryption Standard (DES) and the Triple Data Encryption Algorithm (TDEA) as defined in U.S. Federal Information Processing Standards Publication 46-3.
<u>dsa</u>	Package dsa implements the Digital Signature Algorithm, as defined in FIPS 186-3.
<u>ecdsa</u>	Package ecdsa implements the Elliptic Curve Digital Signature Algorithm, as defined in FIPS 186-3.
<u>elliptic</u>	Package elliptic implements several standard elliptic curves over prime fields.
<u>hmac</u>	Package hmac implements the Keyed-Hash Message Authentication Code (HMAC) as defined in U.S. Federal Information Processing Standards Publication 198.
<u>md5</u>	Package md5 implements the MD5 hash algorithm as defined in RFC 1321.
<u>rand</u>	Package rand implements a cryptographically secure pseudorandom number generator.
<u>rc4</u>	Package rc4 implements RC4 encryption, as defined in Bruce Schneier's Applied Cryptography.
<u>rsa</u>	Package rsa implements RSA encryption as specified in PKCS#1.
<u>sha1</u>	Package sha1 implements the SHA1 hash algorithm as defined in RFC 3174.
<u>sha256</u>	Package sha256 implements the SHA224 and SHA256 hash algorithms as defined in FIPS 180-2.
<u>sha512</u>	Package sha512 implements the SHA384 and SHA512 hash algorithms as defined in FIPS 180-2.
<u>subtle</u>	Package subtle implements functions that are often useful in cryptographic code but require careful thought to use correctly.

tls	Package <code>tls</code> partially implements TLS 1.0, as specified in RFC 2246.
x509	Package <code>x509</code> parses X.509-encoded keys and certificates.
pkix	Package <code>pkix</code> contains shared, low level structures used for ASN.1 parsing and serialization of X.509 certificates, CRL and OCSP.
database	
sql	Package <code>sql</code> provides a generic interface around SQL (or SQL-like) databases.
driver	Package <code>driver</code> defines interfaces to be implemented by database drivers as used by package <code>sql</code> .
debug	
dwarf	Package <code>dwarf</code> provides access to DWARF debugging information loaded from executable files, as defined in the DWARF 2.0 Standard at http://dwarfstd.org/doc/dwarf-2.0.0.pdf
elf	Package <code>elf</code> implements access to ELF object files.
gosym	Package <code>gosym</code> implements access to the Go symbol and line number tables embedded in Go binaries generated by the <code>gc</code> compilers.
macho	Package <code>macho</code> implements access to Mach-O object files.
pe	Package <code>pe</code> implements access to PE (Microsoft Windows Portable Executable) files.
encoding	
ascii85	Package <code>ascii85</code> implements the <code>ascii85</code> data encoding as used in the <code>btoa</code> tool and Adobe's PostScript and PDF document formats.
asn1	Package <code>asn1</code> implements parsing of DER-encoded ASN.1 data structures, as defined in ITU-T Rec X.690.
base32	Package <code>base32</code> implements <code>base32</code> encoding as specified by RFC 4648.
base64	Package <code>base64</code> implements <code>base64</code> encoding as specified by RFC 4648.
binary	Package <code>binary</code> implements translation between numbers and byte sequences and encoding and decoding of varints.

<u>csv</u>	Package csv reads and writes comma-separated values (CSV) files.
<u>gob</u>	Package gob manages streams of gobs - binary values exchanged between an Encoder (transmitter) and a Decoder (receiver).
<u>hex</u>	Package hex implements hexadecimal encoding and decoding.
<u>json</u>	Package json implements encoding and decoding of JSON objects as defined in RFC 4627.
<u>pem</u>	Package pem implements the PEM data encoding, which originated in Privacy Enhanced Mail.
<u>xml</u>	Package xml implements a simple XML 1.0 parser that understands XML name spaces.
<u>errors</u>	Package errors implements functions to manipulate errors.
<u>expvar</u>	Package expvar provides a standardized interface to public variables, such as operation counters in servers.
<u>flag</u>	Package flag implements command-line flag parsing.
<u>fmt</u>	Package fmt implements formatted I/O with functions analogous to C's printf and scanf.
<u>go</u>	
<u>ast</u>	Package ast declares the types used to represent syntax trees for Go packages.
<u>build</u>	Package build gathers information about Go packages.
<u>doc</u>	Package doc extracts source code documentation from a Go AST.
<u>parser</u>	Package parser implements a parser for Go source files.
<u>printer</u>	Package printer implements printing of AST nodes.
<u>scanner</u>	Package scanner implements a scanner for Go source text.
<u>token</u>	Package token defines constants representing the lexical tokens of the Go programming language and basic operations on tokens (printing, predicates).
<u>hash</u>	Package hash provides interfaces for hash functions.
<u>adler32</u>	Package adler32 implements the Adler-32 checksum.
<u>crc32</u>	Package crc32 implements the 32-bit cyclic redundancy check, or CRC-32, checksum.
	Package crc64 implements the 64-bit cyclic redundancy

crc64	check, or CRC-64, checksum.
fnv	Package fnv implements FNV-1 and FNV-1a, non-cryptographic hash functions created by Glenn Fowler, Landon Curt Noll, and Phong Vo.
html	Package html provides functions for escaping and unescaping HTML text.
template	Package template (html/template) implements data-driven templates for generating HTML output safe against code injection.
image	Package image implements a basic 2-D image library.
color	Package color implements a basic color library.
draw	Package draw provides image composition functions.
gif	Package gif implements a GIF image decoder.
jpeg	Package jpeg implements a JPEG image decoder and encoder.
png	Package png implements a PNG image decoder and encoder.
index	
suffixarray	Package suffixarray implements substring search in logarithmic time using an in-memory suffix array.
io	Package io provides basic interfaces to I/O primitives.
ioutil	Package ioutil implements some I/O utility functions.
log	Package log implements a simple logging package.
syslog	Package syslog provides a simple interface to the system log service.
math	Package math provides basic constants and mathematical functions.
big	Package big implements multi-precision arithmetic (big numbers).
cmplx	Package cmplx provides basic constants and mathematical functions for complex numbers.
rand	Package rand implements pseudo-random number generators.
mime	Package mime implements parts of the MIME spec.
multipart	Package multipart implements MIME multipart parsing, as defined in RFC 2046.
	Package net provides a portable interface for network I/O,

[net](#) including TCP/IP, UDP, domain name resolution, and Unix domain sockets.

[http](#) Package http provides HTTP client and server implementations.

[cgi](#) Package cgi implements CGI (Common Gateway Interface) as specified in RFC 3875.

[fcgi](#) Package fcgi implements the FastCGI protocol.

[httpstest](#) Package httpstest provides utilities for HTTP testing.

[httputil](#) Package httputil provides HTTP utility functions, complementing the more common ones in the net/http package.

[pprof](#) Package pprof serves via its HTTP server runtime profiling data in the format expected by the pprof visualization tool.

[mail](#) Package mail implements parsing of mail messages.

[rpc](#) Package rpc provides access to the exported methods of an object across a network or other I/O connection.

[jsonrpc](#) Package jsonrpc implements a JSON-RPC ClientCodec and ServerCodec for the rpc package.

[smtp](#) Package smtp implements the Simple Mail Transfer Protocol as defined in RFC 5321.

[textproto](#) Package textproto implements generic support for text-based request/response protocols in the style of HTTP, NNTP, and SMTP.

[url](#) Package url parses URLs and implements query escaping.

[os](#) Package os provides a platform-independent interface to operating system functionality.

[exec](#) Package exec runs external commands.

[signal](#) Package signal implements access to incoming signals.

[user](#) Package user allows user account lookups by name or id.

[path](#) Package path implements utility routines for manipulating slash-separated paths.

[filepath](#) Package filepath implements utility routines for manipulating filename paths in a way compatible with the target operating system-defined file paths.

[reflect](#) Package reflect implements run-time reflection, allowing a

program to manipulate objects with arbitrary types.

[regexp](#)

Package regexp implements regular expression search.

[syntax](#)

Package syntax parses regular expressions into parse trees and compiles parse trees into programs.

[runtime](#)

Package runtime contains operations that interact with Go's runtime system, such as functions to control goroutines.

[cgo](#)

Package cgo contains runtime support for code generated by the cgo tool.

[debug](#)

Package debug contains facilities for programs to debug themselves while they are running.

[pprof](#)

Package pprof writes runtime profiling data in the format expected by the pprof visualization tool.

[sort](#)

Package sort provides primitives for sorting slices and user-defined collections.

[strconv](#)

Package strconv implements conversions to and from string representations of basic data types.

[strings](#)

Package strings implements simple functions to manipulate strings.

[sync](#)

Package sync provides basic synchronization primitives such as mutual exclusion locks.

[atomic](#)

Package atomic provides low-level atomic memory primitives useful for implementing synchronization algorithms.

[syscall](#)

Package syscall contains an interface to the low-level operating system primitives.

[testing](#)

Package testing provides support for automated testing of Go packages.

[iotest](#)

Package iotest implements Readers and Writers useful mainly for testing.

[quick](#)

Package quick implements utility functions to help with black box testing.

[text](#)

[scanner](#)

Package scanner provides a scanner and tokenizer for UTF-8-encoded text.

[tabwriter](#)

Package tabwriter implements a write filter (tabwriter.Writer) that translates tabbed columns in input into properly aligned

	text.
template	Package template implements data-driven templates for generating textual output.
parse	Package parse builds parse trees for templates as defined by text/template and html/template.
time	Package time provides functionality for measuring and displaying time.
unicode	Package unicode provides data and functions to test some properties of Unicode code points.
utf16	Package utf16 implements encoding and decoding of UTF-16 sequences.
utf8	Package utf8 implements functions and constants to support text encoded in UTF-8.
unsafe	Package unsafe contains operations that step around the type safety of Go programs.

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg/archive

Name	Synopsis
------	----------

tar	Package tar implements access to tar archives.
---------------------	--

zip	Package zip provides support for reading and writing ZIP archives.
---------------------	--

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package tar

```
import "archive/tar"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package tar implements access to tar archives. It aims to cover most of the variations, including those produced by GNU and BSD tars.

References:

<http://www.freebsd.org/cgi/man.cgi?query=tar&sektion=5>

http://www.gnu.org/software/tar/manual/html_node/Standard.html

Index

[Constants](#)

[Variables](#)

[type Header](#)

[type Reader](#)

[func NewReader\(r io.Reader\) *Reader](#)

[func \(tr *Reader\) Next\(\) \(*Header, error\)](#)

[func \(tr *Reader\) Read\(b \[\]byte\) \(n int, err error\)](#)

[type Writer](#)

[func NewWriter\(w io.Writer\) *Writer](#)

[func \(tw *Writer\) Close\(\) error](#)

[func \(tw *Writer\) Flush\(\) error](#)

[func \(tw *Writer\) Write\(b \[\]byte\) \(n int, err error\)](#)

[func \(tw *Writer\) WriteHeader\(hdr *Header\) error](#)

Package files

[common.go](#) [reader.go](#) [writer.go](#)

Constants

```
const (  
    // Types  
    TypeReg          = '0'    // regular file  
    TypeRegA        = '\x00' // regular file  
    TypeLink        = '1'    // hard link  
    TypeSymlink     = '2'    // symbolic link  
    TypeChar        = '3'    // character device node  
    TypeBlock       = '4'    // block device node  
    TypeDir         = '5'    // directory  
    TypeFifo        = '6'    // fifo node  
    TypeCont        = '7'    // reserved  
    TypeXHeader     = 'x'    // extended header  
    TypeXGlobalHeader = 'g'  // global extended header  
)
```

Variables

```
var (  
    ErrWriteTooLong      = errors.New("archive/tar: write too long")  
    ErrFieldTooLong      = errors.New("archive/tar: header field too l  
    ErrWriteAfterClose = errors.New("archive/tar: write after close"  
)
```

```
var (  
    ErrHeader = errors.New("archive/tar: invalid tar header")  
)
```

type [Header](#)

```
type Header struct {
    Name      string // name of header file entry
    Mode      int64  // permission and mode bits
    Uid       int    // user id of owner
    Gid       int    // group id of owner
    Size      int64  // length in bytes
    ModTime   time.Time // modified time
    Typeflag  byte   // type of header entry
    Linkname  string // target name of link
    Uname     string // user name of owner
    Gname     string // group name of owner
    Devmajor  int64  // major number of character or block device
    Devminor  int64  // minor number of character or block device
    AccessTime time.Time // access time
    ChangeTime time.Time // status change time
}
```

A Header represents a single header in a tar archive. Some fields may not be populated.

type [Reader](#)

```
type Reader struct {  
    // contains filtered or unexported fields  
}
```

A Reader provides sequential access to the contents of a tar archive. A tar archive consists of a sequence of files. The Next method advances to the next file in the archive (including the first), and then it can be treated as an io.Reader to access the file's data.

Example:

```
tr := tar.NewReader(r)  
for {  
    hdr, err := tr.Next()  
    if err == io.EOF {  
        // end of tar archive  
        break  
    }  
    if err != nil {  
        // handle error  
    }  
    io.Copy(data, tr)  
}
```

func [NewReader](#)

```
func NewReader(r io.Reader) *Reader
```

NewReader creates a new Reader reading from r.

func (*Reader) [Next](#)

```
func (tr *Reader) Next() (*Header, error)
```

Next advances to the next entry in the tar archive.

func (*Reader) [Read](#)

```
func (tr *Reader) Read(b []byte) (n int, err error)
```

Read reads from the current entry in the tar archive. It returns 0, io.EOF when it reaches the end of that entry, until Next is called to advance to the next entry.

type [Writer](#)

```
type Writer struct {  
    // contains filtered or unexported fields  
}
```

A `Writer` provides sequential writing of a tar archive in POSIX.1 format. A tar archive consists of a sequence of files. Call `WriteHeader` to begin a new file, and then call `Write` to supply that file's data, writing at most `hdr.Size` bytes in total.

Example:

```
tw := tar.NewWriter(w)  
hdr := new(Header)  
hdr.Size = length of data in bytes  
// populate other hdr fields as desired  
if err := tw.WriteHeader(hdr); err != nil {  
    // handle error  
}  
io.Copy(tw, data)  
tw.Close()
```

func [NewWriter](#)

```
func NewWriter(w io.Writer) *Writer
```

`NewWriter` creates a new `Writer` writing to `w`.

func ([*Writer](#)) [Close](#)

```
func (tw *Writer) Close() error
```

`Close` closes the tar archive, flushing any unwritten data to the underlying writer.

func ([*Writer](#)) [Flush](#)

```
func (tw *Writer) Flush() error
```

`Flush` finishes writing the current file (optional).

func ([*Writer](#)) [Write](#)

```
func (tw *Writer) Write(b []byte) (n int, err error)
```

Write writes to the current entry in the tar archive. Write returns the error ErrWriteTooLong if more than hdr.Size bytes are written after WriteHeader.

func (*Writer) [WriteHeader](#)

```
func (tw *Writer) WriteHeader(hdr *Header) error
```

WriteHeader writes hdr and prepares to accept the file's contents. WriteHeader calls Flush if it is not the first header. Calling after a Close will return ErrWriteAfterClose.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package zip

```
import "archive/zip"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package zip provides support for reading and writing ZIP archives.

See: <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>

This package does not support ZIP64 or disk spanning.

Index

[Constants](#)

[Variables](#)

[type File](#)

[func \(f *File\) Open\(\) \(rc io.ReadCloser, err error\)](#)

[type FileHeader](#)

[func FileInfoHeader\(fi os.FileInfo\) \(*FileHeader, error\)](#)

[func \(h *FileHeader\) FileInfo\(\) os.FileInfo](#)

[func \(h *FileHeader\) ModTime\(\) time.Time](#)

[func \(h *FileHeader\) Mode\(\) \(mode os.FileMode\)](#)

[func \(h *FileHeader\) SetModTime\(t time.Time\)](#)

[func \(h *FileHeader\) SetMode\(mode os.FileMode\)](#)

[type ReadCloser](#)

[func OpenReader\(name string\) \(*ReadCloser, error\)](#)

[func \(rc *ReadCloser\) Close\(\) error](#)

[type Reader](#)

[func NewReader\(r io.ReaderAt, size int64\) \(*Reader, error\)](#)

[type Writer](#)

[func NewWriter\(w io.Writer\) *Writer](#)

[func \(w *Writer\) Close\(\) error](#)

[func \(w *Writer\) Create\(name string\) \(io.Writer, error\)](#)

[func \(w *Writer\) CreateHeader\(fh *FileHeader\) \(io.Writer, error\)](#)

Examples

[Reader](#)

[Writer](#)

Package files

[reader.go](#) [struct.go](#) [writer.go](#)

Constants

```
const (  
    Store    uint16 = 0  
    Deflate  uint16 = 8  
)
```

Compression methods.

Variables

```
var (  
    ErrFormat      = errors.New("zip: not a valid zip file")  
    ErrAlgorithm   = errors.New("zip: unsupported compression algorithm")  
    ErrChecksum    = errors.New("zip: checksum error")  
)
```

type [File](#)

```
type File struct {  
    FileHeader  
    // contains filtered or unexported fields  
}
```

func (*File) [Open](#)

```
func (f *File) Open() (rc io.ReadCloser, err error)
```

Open returns a ReadCloser that provides access to the File's contents. Multiple files may be read concurrently.

type [FileHeader](#)

```
type FileHeader struct {
    Name            string
    CreatorVersion  uint16
    ReaderVersion   uint16
    Flags           uint16
    Method          uint16
    ModifiedTime    uint16 // MS-DOS time
    ModifiedDate    uint16 // MS-DOS date
    CRC32           uint32
    CompressedSize  uint32
    UncompressedSize uint32
    Extra           []byte
    ExternalAttrs   uint32 // Meaning depends on CreatorVersion
    Comment         string
}
```

func [FileInfoHeader](#)

```
func FileInfoHeader(fi os.FileInfo) (*FileHeader, error)
```

FileInfoHeader creates a partially-populated FileHeader from an os.FileInfo.

func (*FileHeader) [FileInfo](#)

```
func (h *FileHeader) FileInfo() os.FileInfo
```

FileInfo returns an os.FileInfo for the FileHeader.

func (*FileHeader) [ModTime](#)

```
func (h *FileHeader) ModTime() time.Time
```

ModTime returns the modification time. The resolution is 2s.

func (*FileHeader) [Mode](#)

```
func (h *FileHeader) Mode() (mode os.FileMode)
```

Mode returns the permission and mode bits for the FileHeader.

func (*FileHeader) [SetModTime](#)

```
func (h *FileHeader) SetModTime(t time.Time)
```

SetModTime sets the ModifiedTime and ModifiedDate fields to the given time. The resolution is 2s.

func (*FileHeader) [SetMode](#)

```
func (h *FileHeader) SetMode(mode os.FileMode)
```

SetMode changes the permission and mode bits for the FileHeader.

type [ReadCloser](#)

```
type ReadCloser struct {  
    Reader  
    // contains filtered or unexported fields  
}
```

func [OpenReader](#)

```
func OpenReader(name string) (*ReadCloser, error)
```

OpenReader will open the Zip file specified by name and return a ReadCloser.

func (*ReadCloser) [Close](#)

```
func (rc *ReadCloser) Close() error
```

Close closes the Zip file, rendering it unusable for I/O.

type [Reader](#)

```
type Reader struct {
    File    []*File
    Comment string
    // contains filtered or unexported fields
}
```

? Example

? Example

Code:

```
// Open a zip archive for reading.
r, err := zip.OpenReader("testdata/readme.zip")
if err != nil {
    log.Fatal(err)
}
defer r.Close()

// Iterate through the files in the archive,
// printing some of their contents.
for _, f := range r.File {
    fmt.Printf("Contents of %s:\n", f.Name)
    rc, err := f.Open()
    if err != nil {
        log.Fatal(err)
    }
    _, err = io.CopyN(os.Stdout, rc, 68)
    if err != nil {
        log.Fatal(err)
    }
    rc.Close()
    fmt.Println()
}
```

Output:

```
Contents of README:
This is the source code repository for the Go programming language.
```

func [NewReader](#)

```
func NewReader(r io.ReaderAt, size int64) (*Reader, error)
```

NewReader returns a new Reader reading from r, which is assumed to have the given size in bytes.

type [Writer](#)

```
type Writer struct {  
    // contains filtered or unexported fields  
}
```

Writer implements a zip file writer.

? Example

? Example

Code:

```
// Create a buffer to write our archive to.  
buf := new(bytes.Buffer)  
  
// Create a new zip archive.  
w := zip.NewWriter(buf)  
  
// Add some files to the archive.  
var files = []struct {  
    Name, Body string  
}{  
    {"readme.txt", "This archive contains some text files."},  
    {"gopher.txt", "Gopher names:\nGeorge\nGeoffrey\nGonzo"},  
    {"todo.txt", "Get animal handling licence.\nWrite more examples."  
}  
for _, file := range files {  
    f, err := w.Create(file.Name)  
    if err != nil {  
        log.Fatal(err)  
    }  
    _, err = f.Write([]byte(file.Body))  
    if err != nil {  
        log.Fatal(err)  
    }  
}  
  
// Make sure to check the error on Close.  
err := w.Close()  
if err != nil {  
    log.Fatal(err)  
}
```

func [NewWriter](#)

```
func NewWriter(w io.Writer) *Writer
```

NewWriter returns a new Writer writing a zip file to w.

func (*Writer) [Close](#)

```
func (w *Writer) Close() error
```

Close finishes writing the zip file by writing the central directory. It does not (and can not) close the underlying writer.

func (*Writer) [Create](#)

```
func (w *Writer) Create(name string) (io.Writer, error)
```

Create adds a file to the zip file using the provided name. It returns a Writer to which the file contents should be written. The file's contents must be written to the io.Writer before the next call to Create, CreateHeader, or Close.

func (*Writer) [CreateHeader](#)

```
func (w *Writer) CreateHeader(fh *FileHeader) (io.Writer, error)
```

CreateHeader adds a file to the zip file using the provided FileHeader for the file metadata. It returns a Writer to which the file contents should be written. The file's contents must be written to the io.Writer before the next call to Create, CreateHeader, or Close.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package bufio

```
import "bufio"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `bufio` implements buffered I/O. It wraps an `io.Reader` or `io.Writer` object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.

Index

Variables

type ReadWriter

[func NewReadWriter\(r *Reader, w *Writer\) *ReadWriter](#)

type Reader

[func NewReader\(rd io.Reader\) *Reader](#)

[func NewReaderSize\(rd io.Reader, size int\) *Reader](#)

[func \(b *Reader\) Buffered\(\) int](#)

[func \(b *Reader\) Peek\(n int\) \(\[\]byte, error\)](#)

[func \(b *Reader\) Read\(p \[\]byte\) \(n int, err error\)](#)

[func \(b *Reader\) ReadByte\(\) \(c byte, err error\)](#)

[func \(b *Reader\) ReadBytes\(delim byte\) \(line \[\]byte, err error\)](#)

[func \(b *Reader\) ReadLine\(\) \(line \[\]byte, isPrefix bool, err error\)](#)

[func \(b *Reader\) ReadRune\(\) \(r rune, size int, err error\)](#)

[func \(b *Reader\) ReadSlice\(delim byte\) \(line \[\]byte, err error\)](#)

[func \(b *Reader\) ReadString\(delim byte\) \(line string, err error\)](#)

[func \(b *Reader\) UnreadByte\(\) error](#)

[func \(b *Reader\) UnreadRune\(\) error](#)

type Writer

[func NewWriter\(wr io.Writer\) *Writer](#)

[func NewWriterSize\(wr io.Writer, size int\) *Writer](#)

[func \(b *Writer\) Available\(\) int](#)

[func \(b *Writer\) Buffered\(\) int](#)

[func \(b *Writer\) Flush\(\) error](#)

[func \(b *Writer\) Write\(p \[\]byte\) \(nn int, err error\)](#)

[func \(b *Writer\) WriteByte\(c byte\) error](#)

[func \(b *Writer\) WriteRune\(r rune\) \(size int, err error\)](#)

[func \(b *Writer\) WriteString\(s string\) \(int, error\)](#)

Package files

bufio.go

Variables

```
var (  
    ErrInvalidUnreadByte = errors.New("bufio: invalid use of UnreadB  
    ErrInvalidUnreadRune = errors.New("bufio: invalid use of UnreadR  
    ErrBufferFull        = errors.New("bufio: buffer full")  
    ErrNegativeCount     = errors.New("bufio: negative count")  
)
```

type [ReadWrite](#)

```
type ReadWriter struct {  
    *Reader  
    *Writer  
}
```

ReadWrite stores pointers to a Reader and a Writer. It implements `io.ReadWriter`.

func [NewReadWrite](#)

```
func NewReadWrite(r *Reader, w *Writer) *ReadWrite
```

NewReadWrite allocates a new ReadWriter that dispatches to r and w.

type [Reader](#)

```
type Reader struct {  
    // contains filtered or unexported fields  
}
```

Reader implements buffering for an `io.Reader` object.

func [NewReader](#)

```
func NewReader(rd io.Reader) *Reader
```

`NewReader` returns a new `Reader` whose buffer has the default size.

func [NewReaderSize](#)

```
func NewReaderSize(rd io.Reader, size int) *Reader
```

`NewReaderSize` returns a new `Reader` whose buffer has at least the specified size. If the argument `io.Reader` is already a `Reader` with large enough size, it returns the underlying `Reader`.

func (***Reader**) [Buffered](#)

```
func (b *Reader) Buffered() int
```

`Buffered` returns the number of bytes that can be read from the current buffer.

func (***Reader**) [Peek](#)

```
func (b *Reader) Peek(n int) ([]byte, error)
```

`Peek` returns the next `n` bytes without advancing the reader. The bytes stop being valid at the next read call. If `Peek` returns fewer than `n` bytes, it also returns an error explaining why the read is short. The error is `ErrBufferFull` if `n` is larger than `b`'s buffer size.

func (***Reader**) [Read](#)

```
func (b *Reader) Read(p []byte) (n int, err error)
```

Read reads data into p. It returns the number of bytes read into p. It calls Read at most once on the underlying Reader, hence n may be less than len(p). At EOF, the count will be zero and err will be io.EOF.

func (*Reader) [ReadByte](#)

```
func (b *Reader) ReadByte() (c byte, err error)
```

ReadByte reads and returns a single byte. If no byte is available, returns an error.

func (*Reader) [ReadBytes](#)

```
func (b *Reader) ReadBytes(delim byte) (line []byte, err error)
```

ReadBytes reads until the first occurrence of delim in the input, returning a slice containing the data up to and including the delimiter. If ReadBytes encounters an error before finding a delimiter, it returns the data read before the error and the error itself (often io.EOF). ReadBytes returns err != nil if and only if the returned data does not end in delim.

func (*Reader) [ReadLine](#)

```
func (b *Reader) ReadLine() (line []byte, isPrefix bool, err error)
```

ReadLine tries to return a single line, not including the end-of-line bytes. If the line was too long for the buffer then isPrefix is set and the beginning of the line is returned. The rest of the line will be returned from future calls. isPrefix will be false when returning the last fragment of the line. The returned buffer is only valid until the next call to ReadLine. ReadLine either returns a non-nil line or it returns an error, never both.

func (*Reader) [ReadRune](#)

```
func (b *Reader) ReadRune() (r rune, size int, err error)
```

ReadRune reads a single UTF-8 encoded Unicode character and returns the rune and its size in bytes. If the encoded rune is invalid, it consumes one byte and returns unicode.ReplacementChar (U+FFFD) with a size of 1.

func (*Reader) [ReadSlice](#)

```
func (b *Reader) ReadSlice(delim byte) (line []byte, err error)
```

ReadSlice reads until the first occurrence of delim in the input, returning a slice pointing at the bytes in the buffer. The bytes stop being valid at the next read call. If ReadSlice encounters an error before finding a delimiter, it returns all the data in the buffer and the error itself (often io.EOF). ReadSlice fails with error ErrBufferFull if the buffer fills without a delim. Because the data returned from ReadSlice will be overwritten by the next I/O operation, most clients should use ReadBytes or ReadString instead. ReadSlice returns err != nil if and only if line does not end in delim.

func (*Reader) [ReadString](#)

```
func (b *Reader) ReadString(delim byte) (line string, err error)
```

ReadString reads until the first occurrence of delim in the input, returning a string containing the data up to and including the delimiter. If ReadString encounters an error before finding a delimiter, it returns the data read before the error and the error itself (often io.EOF). ReadString returns err != nil if and only if the returned data does not end in delim.

func (*Reader) [UnreadByte](#)

```
func (b *Reader) UnreadByte() error
```

UnreadByte unreads the last byte. Only the most recently read byte can be unread.

func (*Reader) [UnreadRune](#)

```
func (b *Reader) UnreadRune() error
```

UnreadRune unreads the last rune. If the most recent read operation on the buffer was not a ReadRune, UnreadRune returns an error. (In this regard it is stricter than UnreadByte, which will unread the last byte from any read operation.)

type [Writer](#)

```
type Writer struct {  
    // contains filtered or unexported fields  
}
```

Writer implements buffering for an `io.Writer` object. If an error occurs writing to a Writer, no more data will be accepted and all subsequent writes will return the error.

func [NewWriter](#)

```
func NewWriter(wr io.Writer) *Writer
```

NewWriter returns a new Writer whose buffer has the default size.

func [NewWriterSize](#)

```
func NewWriterSize(wr io.Writer, size int) *Writer
```

NewWriterSize returns a new Writer whose buffer has at least the specified size. If the argument `io.Writer` is already a Writer with large enough size, it returns the underlying Writer.

func (***Writer**) [Available](#)

```
func (w *Writer) Available() int
```

Available returns how many bytes are unused in the buffer.

func (***Writer**) [Buffered](#)

```
func (w *Writer) Buffered() int
```

Buffered returns the number of bytes that have been written into the current buffer.

func (***Writer**) [Flush](#)

```
func (b *Writer) Flush() error
```

Flush writes any buffered data to the underlying io.Writer.

func (*Writer) [Write](#)

```
func (b *Writer) Write(p []byte) (nn int, err error)
```

Write writes the contents of p into the buffer. It returns the number of bytes written. If nn < len(p), it also returns an error explaining why the write is short.

func (*Writer) [WriteByte](#)

```
func (b *Writer) WriteByte(c byte) error
```

WriteByte writes a single byte.

func (*Writer) [WriteRune](#)

```
func (b *Writer) WriteRune(r rune) (size int, err error)
```

WriteRune writes a single Unicode code point, returning the number of bytes written and any error.

func (*Writer) [WriteString](#)

```
func (b *Writer) WriteString(s string) (int, error)
```

WriteString writes a string. It returns the number of bytes written. If the count is less than len(s), it also returns an error explaining why the write is short.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package builtin

```
import "builtin"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package builtin provides documentation for Go's predeclared identifiers. The items documented here are not actually in package builtin but their descriptions here allow godoc to present documentation for the language's special identifiers.

Index

[func append\(slice \[\]Type, elems ...Type\) \[\]Type](#)

[func close\(c chan<- Type\)](#)

[func delete\(m map\[Type\]Type1, key Type\)](#)

[func panic\(v interface{}\)](#)

[func recover\(\) interface{}](#)

[type ComplexType](#)

[func complex\(r, i FloatType\) ComplexType](#)

[type FloatType](#)

[func imag\(c ComplexType\) FloatType](#)

[func real\(c ComplexType\) FloatType](#)

[type IntegerType](#)

[type Type](#)

[func make\(Type, size IntegerType\) Type](#)

[func new\(Type\) *Type](#)

[type Type1](#)

[type bool](#)

[type byte](#)

[type complex128](#)

[type complex64](#)

[type error](#)

[type float32](#)

[type float64](#)

[type int](#)

[func cap\(v Type\) int](#)

[func copy\(dst, src \[\]Type\) int](#)

[func len\(v Type\) int](#)

[type int16](#)

[type int32](#)

[type int64](#)

[type int8](#)

[type rune](#)

[type string](#)

[type uint](#)

[type uint16](#)

[type uint32](#)

[type uint64](#)
[type uint8](#)
[type uintptr](#)

Package files

builtin.go

func [append](#)

```
func append(slice []Type, elems ...Type) []Type
```

The `append` built-in function appends elements to the end of a slice. If it has sufficient capacity, the destination is resliced to accommodate the new elements. If it does not, a new underlying array will be allocated. `Append` returns the updated slice. It is therefore necessary to store the result of `append`, often in the variable holding the slice itself:

```
slice = append(slice, elem1, elem2)  
slice = append(slice, anotherSlice...)
```

func [close](#)

```
func close(c chan<- Type)
```

The `close` built-in function closes a channel, which must be either bidirectional or send-only. It should be executed only by the sender, never the receiver, and has the effect of shutting down the channel after the last sent value is received. After the last value has been received from a closed channel `c`, any receive from `c` will succeed without blocking, returning the zero value for the channel element. The form

```
x, ok := <-c
```

will also set `ok` to `false` for a closed channel.

func [delete](#)

```
func delete(m map[Type]Type1, key Type)
```

The delete built-in function deletes the element with the specified key (m[key]) from the map. If there is no such element, delete is a no-op. If m is nil, delete panics.

func [panic](#)

```
func panic(v interface{})
```

The panic built-in function stops normal execution of the current goroutine. When a function F calls panic, normal execution of F stops immediately. Any functions whose execution was deferred by F are run in the usual way, and then F returns to its caller. To the caller G, the invocation of F then behaves like a call to panic, terminating G's execution and running any deferred functions. This continues until all functions in the executing goroutine have stopped, in reverse order. At that point, the program is terminated and the error condition is reported, including the value of the argument to panic. This termination sequence is called panicking and can be controlled by the built-in function recover.

func [recover](#)

```
func recover() interface{}
```

The `recover` built-in function allows a program to manage behavior of a panicking goroutine. Executing a call to `recover` inside a deferred function (but not any function called by it) stops the panicking sequence by restoring normal execution and retrieves the error value passed to the call of `panic`. If `recover` is called outside the deferred function it will not stop a panicking sequence. In this case, or when the goroutine is not panicking, or if the argument supplied to `panic` was `nil`, `recover` returns `nil`. Thus the return value from `recover` reports whether the goroutine is panicking.

type [ComplexType](#)

```
type ComplexType complex64
```

ComplexType is here for the purposes of documentation only. It is a stand-in for either complex type: complex64 or complex128.

func [complex](#)

```
func complex(r, i FloatType) ComplexType
```

The complex built-in function constructs a complex value from two floating-point values. The real and imaginary parts must be of the same size, either float32 or float64 (or assignable to them), and the return value will be the corresponding complex type (complex64 for float32, complex128 for float64).

type [FloatType](#)

```
type FloatType float32
```

FloatType is here for the purposes of documentation only. It is a stand-in for either float type: float32 or float64.

func [imag](#)

```
func imag(c ComplexType) FloatType
```

The imag built-in function returns the imaginary part of the complex number c. The return value will be floating point type corresponding to the type of c.

func [real](#)

```
func real(c ComplexType) FloatType
```

The real built-in function returns the real part of the complex number c. The return value will be floating point type corresponding to the type of c.

type [IntegerType](#)

```
type IntegerType int
```

IntegerType is here for the purposes of documentation only. It is a stand-in for any integer type: int, uint, int8 etc.

type [Type](#)

```
type Type int
```

Type is here for the purposes of documentation only. It is a stand-in for any Go type, but represents the same type for any given function invocation.

func [make](#)

```
func make(Type, size IntegerType) Type
```

The make built-in function allocates and initializes an object of type slice, map, or chan (only). Like new, the first argument is a type, not a value. Unlike new, make's return type is the same as the type of its argument, not a pointer to it. The specification of the result depends on the type:

Slice: The size specifies the length. The capacity of the slice is equal to its length. A second integer argument may be provided to specify a different capacity; it must be no smaller than the length, so `make([]int, 0, 10)` allocates a slice of length 0 and capacity 10.

Map: An initial allocation is made according to the size but the resulting map has length 0. The size may be omitted, in which case a small starting size is allocated.

Channel: The channel's buffer is initialized with the specified buffer capacity. If zero, or the size is omitted, the channel is unbuffered.

func [new](#)

```
func new(Type) *Type
```

The new built-in function allocates memory. The first argument is a type, not a value, and the value returned is a pointer to a newly allocated zero value of that type.

type [Type1](#)

```
type Type1 int
```

Type1 is here for the purposes of documentation only. It is a stand-in for any Go type, but represents the same type for any given function invocation.

type [bool](#)

```
type bool bool
```

bool is the set of boolean values, true and false.

type [byte](#)

type byte byte

byte is an alias for uint8 and is equivalent to uint8 in all ways. It is used, by convention, to distinguish byte values from 8-bit unsigned integer values.

type [complex128](#)

type complex128 complex128

complex128 is the set of all complex numbers with float64 real and imaginary parts.

type [complex64](#)

```
type complex64 complex64
```

complex64 is the set of all complex numbers with float32 real and imaginary parts.

type [error](#)

```
type error interface {  
    Error() string  
}
```

The error built-in interface type is the conventional interface for representing an error condition, with the nil value representing no error.

type [float32](#)

type float32 float32

float32 is the set of all IEEE-754 32-bit floating-point numbers.

type [float64](#)

type float64 float64

float64 is the set of all IEEE-754 64-bit floating-point numbers.

type [int](#)

```
type int int
```

int is a signed integer type that is at least 32 bits in size. It is a distinct type, however, and not an alias for, say, int32.

func [cap](#)

```
func cap(v Type) int
```

The cap built-in function returns the capacity of v, according to its type:

Array: the number of elements in v (same as len(v)).
Pointer to array: the number of elements in *v (same as len(v)).
Slice: the maximum length the slice can reach when resliced;
if v is nil, cap(v) is zero.
Channel: the channel buffer capacity, in units of elements;
if v is nil, cap(v) is zero.

func [copy](#)

```
func copy(dst, src []Type) int
```

The copy built-in function copies elements from a source slice into a destination slice. (As a special case, it also will copy bytes from a string to a slice of bytes.) The source and destination may overlap. Copy returns the number of elements copied, which will be the minimum of len(src) and len(dst).

func [len](#)

```
func len(v Type) int
```

The len built-in function returns the length of v, according to its type:

Array: the number of elements in v.
Pointer to array: the number of elements in *v (even if v is nil).
Slice, or map: the number of elements in v; if v is nil, len(v) is zero.
String: the number of bytes in v.
Channel: the number of elements queued (unread) in the channel buffer;
if v is nil, len(v) is zero.

type [int16](#)

```
type int16 int16
```

int16 is the set of all signed 16-bit integers. Range: -32768 through 32767.

type [int32](#)

```
type int32 int32
```

int32 is the set of all signed 32-bit integers. Range: -2147483648 through 2147483647.

type [int64](#)

```
type int64 int64
```

int64 is the set of all signed 64-bit integers. Range: -9223372036854775808 through 9223372036854775807.

type [int8](#)

```
type int8 int8
```

int8 is the set of all signed 8-bit integers. Range: -128 through 127.

type [rune](#)

```
type rune rune
```

rune is an alias for int and is equivalent to int in all ways. It is used, by convention, to distinguish character values from integer values. In a future version of Go, it will change to an alias of int32.

type [string](#)

```
type string string
```

string is the set of all strings of 8-bit bytes, conventionally but not necessarily representing UTF-8-encoded text. A string may be empty, but not nil. Values of string type are immutable.

type [uint](#)

```
type uint uint
```

uint is an unsigned integer type that is at least 32 bits in size. It is a distinct type, however, and not an alias for, say, uint32.

type [uint16](#)

type uint16 uint16

uint16 is the set of all unsigned 16-bit integers. Range: 0 through 65535.

type [uint32](#)

type uint32 uint32

uint32 is the set of all unsigned 32-bit integers. Range: 0 through 4294967295.

type [uint64](#)

type uint64 uint64

uint64 is the set of all unsigned 64-bit integers. Range: 0 through 18446744073709551615.

type [uint8](#)

```
type uint8 uint8
```

uint8 is the set of all unsigned 8-bit integers. Range: 0 through 255.

type [uintptr](#)

```
type uintptr uintptr
```

uintptr is an integer type that is large enough to hold the bit pattern of any pointer.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package bytes

```
import "bytes"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package bytes implements functions for the manipulation of byte slices. It is analogous to the facilities of the strings package.

Index

[Constants](#)

[Variables](#)

[func Compare\(a, b \[\]byte\) int](#)

[func Contains\(b, subslice \[\]byte\) bool](#)

[func Count\(s, sep \[\]byte\) int](#)

[func Equal\(a, b \[\]byte\) bool](#)

[func EqualFold\(s, t \[\]byte\) bool](#)

[func Fields\(s \[\]byte\) \[\]\[\]byte](#)

[func FieldsFunc\(s \[\]byte, f func\(rune\) bool\) \[\]\[\]byte](#)

[func HasPrefix\(s, prefix \[\]byte\) bool](#)

[func HasSuffix\(s, suffix \[\]byte\) bool](#)

[func Index\(s, sep \[\]byte\) int](#)

[func IndexAny\(s \[\]byte, chars string\) int](#)

[func IndexByte\(s \[\]byte, c byte\) int](#)

[func IndexFunc\(s \[\]byte, f func\(r rune\) bool\) int](#)

[func IndexRune\(s \[\]byte, r rune\) int](#)

[func Join\(a \[\]\[\]byte, sep \[\]byte\) \[\]byte](#)

[func LastIndex\(s, sep \[\]byte\) int](#)

[func LastIndexAny\(s \[\]byte, chars string\) int](#)

[func LastIndexFunc\(s \[\]byte, f func\(r rune\) bool\) int](#)

[func Map\(mapping func\(r rune\) rune, s \[\]byte\) \[\]byte](#)

[func Repeat\(b \[\]byte, count int\) \[\]byte](#)

[func Replace\(s, old, new \[\]byte, n int\) \[\]byte](#)

[func Runes\(s \[\]byte\) \[\]rune](#)

[func Split\(s, sep \[\]byte\) \[\]\[\]byte](#)

[func SplitAfter\(s, sep \[\]byte\) \[\]\[\]byte](#)

[func SplitAfterN\(s, sep \[\]byte, n int\) \[\]\[\]byte](#)

[func SplitN\(s, sep \[\]byte, n int\) \[\]\[\]byte](#)

[func Title\(s \[\]byte\) \[\]byte](#)

[func ToLower\(s \[\]byte\) \[\]byte](#)

[func ToLowerSpecial\(_ case unicode.SpecialCase, s \[\]byte\) \[\]byte](#)

[func ToTitle\(s \[\]byte\) \[\]byte](#)

[func ToTitleSpecial\(_ case unicode.SpecialCase, s \[\]byte\) \[\]byte](#)

[func ToUpper\(s \[\]byte\) \[\]byte](#)

[func ToUpperSpecial\(_ case unicode.SpecialCase, s \[\]byte\) \[\]byte](#)

[func Trim\(s \[\]byte, cutset string\) \[\]byte](#)
[func TrimFunc\(s \[\]byte, f func\(r rune\) bool\) \[\]byte](#)
[func TrimLeft\(s \[\]byte, cutset string\) \[\]byte](#)
[func TrimLeftFunc\(s \[\]byte, f func\(r rune\) bool\) \[\]byte](#)
[func TrimRight\(s \[\]byte, cutset string\) \[\]byte](#)
[func TrimRightFunc\(s \[\]byte, f func\(r rune\) bool\) \[\]byte](#)
[func TrimSpace\(s \[\]byte\) \[\]byte](#)
[type Buffer](#)

[func NewBuffer\(buf \[\]byte\) *Buffer](#)
[func NewBufferString\(s string\) *Buffer](#)
[func \(b *Buffer\) Bytes\(\) \[\]byte](#)
[func \(b *Buffer\) Len\(\) int](#)
[func \(b *Buffer\) Next\(n int\) \[\]byte](#)
[func \(b *Buffer\) Read\(p \[\]byte\) \(n int, err error\)](#)
[func \(b *Buffer\) ReadByte\(\) \(c byte, err error\)](#)
[func \(b *Buffer\) ReadBytes\(delim byte\) \(line \[\]byte, err error\)](#)
[func \(b *Buffer\) ReadFrom\(r io.Reader\) \(n int64, err error\)](#)
[func \(b *Buffer\) ReadRune\(\) \(r rune, size int, err error\)](#)
[func \(b *Buffer\) ReadString\(delim byte\) \(line string, err error\)](#)
[func \(b *Buffer\) Reset\(\)](#)
[func \(b *Buffer\) String\(\) string](#)
[func \(b *Buffer\) Truncate\(n int\)](#)
[func \(b *Buffer\) UnreadByte\(\) error](#)
[func \(b *Buffer\) UnreadRune\(\) error](#)
[func \(b *Buffer\) Write\(p \[\]byte\) \(n int, err error\)](#)
[func \(b *Buffer\) WriteByte\(c byte\) error](#)
[func \(b *Buffer\) WriteRune\(r rune\) \(n int, err error\)](#)
[func \(b *Buffer\) WriteString\(s string\) \(n int, err error\)](#)
[func \(b *Buffer\) WriteTo\(w io.Writer\) \(n int64, err error\)](#)

[type Reader](#)

[func NewReader\(b \[\]byte\) *Reader](#)
[func \(r *Reader\) Len\(\) int](#)
[func \(r *Reader\) Read\(b \[\]byte\) \(n int, err error\)](#)
[func \(r *Reader\) ReadAt\(b \[\]byte, off int64\) \(n int, err error\)](#)
[func \(r *Reader\) ReadByte\(\) \(b byte, err error\)](#)
[func \(r *Reader\) ReadRune\(\) \(ch rune, size int, err error\)](#)
[func \(r *Reader\) Seek\(offset int64, whence int\) \(int64, error\)](#)
[func \(r *Reader\) UnreadByte\(\) error](#)
[func \(r *Reader\) UnreadRune\(\) error](#)

[Bugs](#)

Examples

[Buffer](#)

[Buffer \(Reader\)](#)

Package files

[buffer.go](#) [bytes.go](#) [bytes_decl.go](#) [reader.go](#)

Constants

```
const MinRead = 512
```

MinRead is the minimum slice size passed to a Read call by Buffer.ReadFrom. As long as the Buffer has at least MinRead bytes beyond what is required to hold the contents of r, ReadFrom will not grow the underlying buffer.

Variables

```
var ErrTooLarge = errors.New("bytes.Buffer: too large")
```

ErrTooLarge is passed to panic if memory cannot be allocated to store data in a buffer.

func Compare

```
func Compare(a, b []byte) int
```

Compare returns an integer comparing the two byte arrays lexicographically. The result will be 0 if $a==b$, -1 if $a < b$, and +1 if $a > b$. A nil argument is equivalent to an empty slice.

func Contains

```
func Contains(b, subslice []byte) bool
```

Contains returns whether subslice is within b.

func Count

```
func Count(s, sep []byte) int
```

Count counts the number of non-overlapping instances of sep in s.

func Equal

```
func Equal(a, b []byte) bool
```

Equal returns a boolean reporting whether `a == b`. A nil argument is equivalent to an empty slice.

func [EqualFold](#)

```
func EqualFold(s, t []byte) bool
```

EqualFold reports whether s and t, interpreted as UTF-8 strings, are equal under Unicode case-folding.

func [Fields](#)

```
func Fields(s []byte) [][]byte
```

Fields splits the array `s` around each instance of one or more consecutive white space characters, returning a slice of subarrays of `s` or an empty list if `s` contains only white space.

func [FieldsFunc](#)

```
func FieldsFunc(s []byte, f func(rune) bool) [][]byte
```

FieldsFunc interprets `s` as a sequence of UTF-8-encoded Unicode code points. It splits the array `s` at each run of code points `c` satisfying `f(c)` and returns a slice of subarrays of `s`. If no code points in `s` satisfy `f(c)`, an empty slice is returned.

func [HasPrefix](#)

```
func HasPrefix(s, prefix []byte) bool
```

HasPrefix tests whether the byte array `s` begins with `prefix`.

func [HasSuffix](#)

```
func HasSuffix(s, suffix []byte) bool
```

HasSuffix tests whether the byte array s ends with suffix.

func [Index](#)

```
func Index(s, sep []byte) int
```

Index returns the index of the first instance of sep in s, or -1 if sep is not present in s.

func [IndexAny](#)

```
func IndexAny(s []byte, chars string) int
```

IndexAny interprets s as a sequence of UTF-8-encoded Unicode code points. It returns the byte index of the first occurrence in s of any of the Unicode code points in chars. It returns -1 if chars is empty or if there is no code point in common.

func IndexByte

```
func IndexByte(s []byte, c byte) int
```

IndexByte returns the index of the first instance of c in s, or -1 if c is not present in s.

func [IndexFunc](#)

```
func IndexFunc(s []byte, f func(r rune) bool) int
```

IndexFunc interprets s as a sequence of UTF-8-encoded Unicode code points. It returns the byte index in s of the first Unicode code point satisfying f(c), or -1 if none do.

func [IndexRune](#)

```
func IndexRune(s []byte, r rune) int
```

IndexRune interprets s as a sequence of UTF-8-encoded Unicode code points. It returns the byte index of the first occurrence in s of the given rune. It returns -1 if rune is not present in s.

func [Join](#)

```
func Join(a [][]byte, sep []byte) []byte
```

Join concatenates the elements of a to create a single byte array. The separator sep is placed between elements in the resulting array.

func LastIndex

```
func LastIndex(s, sep []byte) int
```

LastIndex returns the index of the last instance of sep in s, or -1 if sep is not present in s.

func [LastIndexAny](#)

```
func LastIndexAny(s []byte, chars string) int
```

LastIndexAny interprets s as a sequence of UTF-8-encoded Unicode code points. It returns the byte index of the last occurrence in s of any of the Unicode code points in chars. It returns -1 if chars is empty or if there is no code point in common.

func [LastIndexFunc](#)

```
func LastIndexFunc(s []byte, f func(r rune) bool) int
```

LastIndexFunc interprets s as a sequence of UTF-8-encoded Unicode code points. It returns the byte index in s of the last Unicode code point satisfying f(c), or -1 if none do.

func [Map](#)

```
func Map(mapping func(r rune) rune, s []byte) []byte
```

Map returns a copy of the byte array `s` with all its characters modified according to the mapping function. If mapping returns a negative value, the character is dropped from the string with no replacement. The characters in `s` and the output are interpreted as UTF-8-encoded Unicode code points.

func Repeat

```
func Repeat(b []byte, count int) []byte
```

Repeat returns a new byte slice consisting of count copies of b.

func [Replace](#)

```
func Replace(s, old, new []byte, n int) []byte
```

Replace returns a copy of the slice `s` with the first `n` non-overlapping instances of `old` replaced by `new`. If `n < 0`, there is no limit on the number of replacements.

func [Runes](#)

```
func Runes(s []byte) []rune
```

Runes returns a slice of runes (Unicode code points) equivalent to s.

func [Split](#)

```
func Split(s, sep []byte) [][]byte
```

Split slices `s` into all subslices separated by `sep` and returns a slice of the subslices between those separators. If `sep` is empty, Split splits after each UTF-8 sequence. It is equivalent to `SplitN` with a count of `-1`.

func [SplitAfter](#)

```
func SplitAfter(s, sep []byte) [][]byte
```

SplitAfter slices `s` into all subslices after each instance of `sep` and returns a slice of those subslices. If `sep` is empty, SplitAfter splits after each UTF-8 sequence. It is equivalent to SplitAfterN with a count of -1.

func [SplitAfterN](#)

```
func SplitAfterN(s, sep []byte, n int) [][]byte
```

SplitAfterN slices s into subslices after each instance of sep and returns a slice of those subslices. If sep is empty, SplitAfterN splits after each UTF-8 sequence.

The count determines the number of subslices to return:

n > 0: at most n subslices; the last subslice will be the unsplit re

n == 0: the result is nil (zero subslices)

n < 0: all subslices

func [SplitN](#)

```
func SplitN(s, sep []byte, n int) [][]byte
```

SplitN slices `s` into subslices separated by `sep` and returns a slice of the subslices between those separators. If `sep` is empty, SplitN splits after each UTF-8 sequence. The count determines the number of subslices to return:

```
n > 0: at most n subslices; the last subslice will be the unsplit re  
n == 0: the result is nil (zero subslices)  
n < 0: all subslices
```

func [Title](#)

```
func Title(s []byte) []byte
```

Title returns a copy of s with all Unicode letters that begin words mapped to their title case.

func ToLower

```
func ToLower(s []byte) []byte
```

ToUpper returns a copy of the byte array s with all Unicode letters mapped to their lower case.

func [ToLowerSpecial](#)

```
func ToLowerSpecial(_case unicode.SpecialCase, s []byte) []byte
```

ToLowerSpecial returns a copy of the byte array `s` with all Unicode letters mapped to their lower case, giving priority to the special casing rules.

func [ToTitle](#)

```
func ToTitle(s []byte) []byte
```

ToTitle returns a copy of the byte array s with all Unicode letters mapped to their title case.

func [ToTitleSpecial](#)

```
func ToTitleSpecial(_case unicode.SpecialCase, s []byte) []byte
```

ToTitleSpecial returns a copy of the byte array `s` with all Unicode letters mapped to their title case, giving priority to the special casing rules.

func ToUpper

```
func ToUpper(s []byte) []byte
```

ToUpper returns a copy of the byte array s with all Unicode letters mapped to their upper case.

func [ToUpperSpecial](#)

```
func ToUpperSpecial(_case unicode.SpecialCase, s []byte) []byte
```

ToUpperSpecial returns a copy of the byte array `s` with all Unicode letters mapped to their upper case, giving priority to the special casing rules.

func [Trim](#)

```
func Trim(s []byte, cutset string) []byte
```

Trim returns a subslice of s by slicing off all leading and trailing UTF-8-encoded Unicode code points contained in cutset.

func [TrimFunc](#)

```
func TrimFunc(s []byte, f func(r rune) bool) []byte
```

TrimFunc returns a subslice of s by slicing off all leading and trailing UTF-8-encoded Unicode code points c that satisfy f(c).

func [TrimLeft](#)

```
func TrimLeft(s []byte, cutset string) []byte
```

TrimLeft returns a subslice of s by slicing off all leading UTF-8-encoded Unicode code points contained in cutset.

func [TrimLeftFunc](#)

```
func TrimLeftFunc(s []byte, f func(r rune) bool) []byte
```

TrimLeftFunc returns a subslice of s by slicing off all leading UTF-8-encoded Unicode code points c that satisfy f(c).

func [TrimRight](#)

```
func TrimRight(s []byte, cutset string) []byte
```

TrimRight returns a subslice of s by slicing off all trailing UTF-8-encoded Unicode code points that are contained in cutset.

func [TrimRightFunc](#)

```
func TrimRightFunc(s []byte, f func(r rune) bool) []byte
```

TrimRightFunc returns a subslice of s by slicing off all trailing UTF-8 encoded Unicode code points c that satisfy f(c).

func [TrimSpace](#)

```
func TrimSpace(s []byte) []byte
```

TrimSpace returns a subslice of s by slicing off all leading and trailing white space, as defined by Unicode.

type [Buffer](#)

```
type Buffer struct {  
    // contains filtered or unexported fields  
}
```

A Buffer is a variable-sized buffer of bytes with Read and Write methods. The zero value for Buffer is an empty buffer ready to use.

? Example

? Example

Code:

```
var b Buffer // A Buffer needs no initialization.  
b.Write([]byte("Hello "))  
b.Write([]byte("world!"))  
b.WriteTo(os.Stdout)
```

Output:

Hello world!

? Example (Reader)

? Example (Reader)

Code:

```
// A Buffer can turn a string or a []byte into an io.Reader.  
buf := NewBufferString("R29waGVycyBydWxlIQ==")  
dec := base64.NewDecoder(base64.StdEncoding, buf)  
io.Copy(os.Stdout, dec)
```

Output:

Gophers rule!

func [NewBuffer](#)

```
func NewBuffer(buf []byte) *Buffer
```

NewBuffer creates and initializes a new Buffer using buf as its initial contents. It is intended to prepare a Buffer to read existing data. It can also be used to size the internal buffer for writing. To do that, buf should have the desired capacity but a length of zero.

In most cases, new(Buffer) (or just declaring a Buffer variable) is sufficient to initialize a Buffer.

func [NewBufferString](#)

```
func NewBufferString(s string) *Buffer
```

NewBufferString creates and initializes a new Buffer using string s as its initial contents. It is intended to prepare a buffer to read an existing string.

In most cases, new(Buffer) (or just declaring a Buffer variable) is sufficient to initialize a Buffer.

func (***Buffer**) [Bytes](#)

```
func (b *Buffer) Bytes() []byte
```

Bytes returns a slice of the contents of the unread portion of the buffer; len(b.Bytes()) == b.Len(). If the caller changes the contents of the returned slice, the contents of the buffer will change provided there are no intervening method calls on the Buffer.

func (***Buffer**) [Len](#)

```
func (b *Buffer) Len() int
```

Len returns the number of bytes of the unread portion of the buffer; b.Len() == len(b.Bytes()).

func (***Buffer**) [Next](#)

```
func (b *Buffer) Next(n int) []byte
```

Next returns a slice containing the next n bytes from the buffer, advancing the buffer as if the bytes had been returned by Read. If there are fewer than n bytes

in the buffer, Next returns the entire buffer. The slice is only valid until the next call to a read or write method.

func (*Buffer) [Read](#)

```
func (b *Buffer) Read(p []byte) (n int, err error)
```

Read reads the next len(p) bytes from the buffer or until the buffer is drained. The return value n is the number of bytes read. If the buffer has no data to return, err is io.EOF (unless len(p) is zero); otherwise it is nil.

func (*Buffer) [ReadByte](#)

```
func (b *Buffer) ReadByte() (c byte, err error)
```

ReadByte reads and returns the next byte from the buffer. If no byte is available, it returns error io.EOF.

func (*Buffer) [ReadBytes](#)

```
func (b *Buffer) ReadBytes(delim byte) (line []byte, err error)
```

ReadBytes reads until the first occurrence of delim in the input, returning a slice containing the data up to and including the delimiter. If ReadBytes encounters an error before finding a delimiter, it returns the data read before the error and the error itself (often io.EOF). ReadBytes returns err != nil if and only if the returned data does not end in delim.

func (*Buffer) [ReadFrom](#)

```
func (b *Buffer) ReadFrom(r io.Reader) (n int64, err error)
```

ReadFrom reads data from r until EOF and appends it to the buffer. The return value n is the number of bytes read. Any error except io.EOF encountered during the read is also returned. If the buffer becomes too large, ReadFrom will panic with ErrTooLarge.

func (*Buffer) [ReadRune](#)

```
func (b *Buffer) ReadRune() (r rune, size int, err error)
```

ReadRune reads and returns the next UTF-8-encoded Unicode code point from the buffer. If no bytes are available, the error returned is io.EOF. If the bytes are an erroneous UTF-8 encoding, it consumes one byte and returns U+FFFD, 1.

func (*Buffer) [ReadString](#)

```
func (b *Buffer) ReadString(delim byte) (line string, err error)
```

ReadString reads until the first occurrence of delim in the input, returning a string containing the data up to and including the delimiter. If ReadString encounters an error before finding a delimiter, it returns the data read before the error and the error itself (often io.EOF). ReadString returns err != nil if and only if the returned data does not end in delim.

func (*Buffer) [Reset](#)

```
func (b *Buffer) Reset()
```

Reset resets the buffer so it has no content. b.Reset() is the same as b.Truncate(0).

func (*Buffer) [String](#)

```
func (b *Buffer) String() string
```

String returns the contents of the unread portion of the buffer as a string. If the Buffer is a nil pointer, it returns "<nil>".

func (*Buffer) [Truncate](#)

```
func (b *Buffer) Truncate(n int)
```

Truncate discards all but the first n unread bytes from the buffer. It panics if n is negative or greater than the length of the buffer.

func (*Buffer) [UnreadByte](#)

```
func (b *Buffer) UnreadByte() error
```

UnreadByte unreads the last byte returned by the most recent read operation. If

write has happened since the last read, `UnreadByte` returns an error.

func (*Buffer) [UnreadRune](#)

```
func (b *Buffer) UnreadRune() error
```

`UnreadRune` unread the last rune returned by `ReadRune`. If the most recent read or write operation on the buffer was not a `ReadRune`, `UnreadRune` returns an error. (In this regard it is stricter than `UnreadByte`, which will unread the last byte from any read operation.)

func (*Buffer) [Write](#)

```
func (b *Buffer) Write(p []byte) (n int, err error)
```

`Write` appends the contents of `p` to the buffer. The return value `n` is the length of `p`; `err` is always `nil`. If the buffer becomes too large, `Write` will panic with `ErrTooLarge`.

func (*Buffer) [WriteByte](#)

```
func (b *Buffer) WriteByte(c byte) error
```

`WriteByte` appends the byte `c` to the buffer. The returned error is always `nil`, but is included to match `bufio.Writer's WriteByte`. If the buffer becomes too large, `WriteByte` will panic with `ErrTooLarge`.

func (*Buffer) [WriteRune](#)

```
func (b *Buffer) WriteRune(r rune) (n int, err error)
```

`WriteRune` appends the UTF-8 encoding of Unicode code point `r` to the buffer, returning its length and an error, which is always `nil` but is included to match `bufio.Writer's WriteRune`. If the buffer becomes too large, `WriteRune` will panic with `ErrTooLarge`.

func (*Buffer) [WriteString](#)

```
func (b *Buffer) WriteString(s string) (n int, err error)
```

WriteString appends the contents of s to the buffer. The return value n is the length of s; err is always nil. If the buffer becomes too large, WriteString will panic with ErrTooLarge.

func (*Buffer) [WriteTo](#)

```
func (b *Buffer) WriteTo(w io.Writer) (n int64, err error)
```

WriteTo writes data to w until the buffer is drained or an error occurs. The return value n is the number of bytes written; it always fits into an int, but it is int64 to match the io.WriterTo interface. Any error encountered during the write is also returned.

type [Reader](#)

```
type Reader struct {  
    // contains filtered or unexported fields  
}
```

A Reader implements the `io.Reader`, `io.ReaderAt`, `io.Seeker`, `io.ByteScanner`, and `io.RuneScanner` interfaces by reading from a byte slice. Unlike a Buffer, a Reader is read-only and supports seeking.

func [NewReader](#)

```
func NewReader(b []byte) *Reader
```

NewReader returns a new Reader reading from b.

func (***Reader**) [Len](#)

```
func (r *Reader) Len() int
```

Len returns the number of bytes of the unread portion of the slice.

func (***Reader**) [Read](#)

```
func (r *Reader) Read(b []byte) (n int, err error)
```

func (***Reader**) [ReadAt](#)

```
func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)
```

func (***Reader**) [ReadByte](#)

```
func (r *Reader) ReadByte() (b byte, err error)
```

func (***Reader**) [ReadRune](#)

```
func (r *Reader) ReadRune() (ch rune, size int, err error)
```

func (***Reader**) [Seek](#)

```
func (r *Reader) Seek(offset int64, whence int) (int64, error)
```

Seek implements the io.Seeker interface.

func (*Reader) [UnreadByte](#)

```
func (r *Reader) UnreadByte() error
```

func (*Reader) [UnreadRune](#)

```
func (r *Reader) UnreadRune() error
```

Bugs

The rule Title uses for word boundaries does not handle Unicode punctuation properly.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg/compress

Name	Synopsis
bzip2	Package bzip2 implements bzip2 decompression.
flate	Package flate implements the DEFLATE compressed data format, described in RFC 1951.
gzip	Package gzip implements reading and writing of gzip format compressed files, as specified in RFC 1952.
lzw	Package lzw implements the Lempel-Ziv-Welch compressed data format, described in T. A. Welch, "A Technique for High-Performance Data Compression", Computer, 17(6) (June 1984), pp 8-19.
zlib	Package zlib implements reading and writing of zlib format compressed data, as specified in RFC 1950.

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package bzip2

```
import "compress/bzip2"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package bzip2 implements bzip2 decompression.

Index

[func NewReader\(r io.Reader\) io.Reader](#)
[type StructuralError](#)
[func \(s StructuralError\) Error\(\) string](#)

Package files

[bit_reader.go](#) [bzip2.go](#) [huffman.go](#) [move_to_front.go](#)

func [NewReader](#)

```
func NewReader(r io.Reader) io.Reader
```

NewReader returns an io.Reader which decompresses bzip2 data from r.

type [StructuralError](#)

```
type StructuralError string
```

A StructuralError is returned when the bzip2 data is found to be syntactically invalid.

func (StructuralError) [Error](#)

```
func (s StructuralError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package flate

```
import "compress/flate"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package flate implements the DEFLATE compressed data format, described in RFC 1951. The gzip and zlib packages implement access to DEFLATE-based file formats.

Index

Constants

[func NewReader\(r io.Reader\) io.ReadCloser](#)

[func NewReaderDict\(r io.Reader, dict \[\]byte\) io.ReadCloser](#)

[type CorruptInputError](#)

[func \(e CorruptInputError\) Error\(\) string](#)

[type InternalError](#)

[func \(e InternalError\) Error\(\) string](#)

[type ReadError](#)

[func \(e *ReadError\) Error\(\) string](#)

[type Reader](#)

[type WriteError](#)

[func \(e *WriteError\) Error\(\) string](#)

[type Writer](#)

[func NewWriter\(w io.Writer, level int\) \(*Writer, error\)](#)

[func NewWriterDict\(w io.Writer, level int, dict \[\]byte\) \(*Writer, error\)](#)

[func \(w *Writer\) Close\(\) error](#)

[func \(w *Writer\) Flush\(\) error](#)

[func \(w *Writer\) Write\(data \[\]byte\) \(n int, err error\)](#)

Package files

[deflate.go](#) [huffman](#) [bit](#) [writer.go](#) [huffman](#) [code.go](#) [inflate.go](#) [reverse](#) [bits.go](#) [token.go](#)

Constants

```
const (  
    NoCompression = 0  
    BestSpeed     = 1  
  
    BestCompression = 9  
    DefaultCompression = -1  
)
```

func [NewReader](#)

```
func NewReader(r io.Reader) io.ReadCloser
```

NewReader returns a new ReadCloser that can be used to read the uncompressed version of r. It is the caller's responsibility to call Close on the ReadCloser when finished reading.

func [NewReaderDict](#)

```
func NewReaderDict(r io.Reader, dict []byte) io.ReadCloser
```

NewReaderDict is like NewReader but initializes the reader with a preset dictionary. The returned Reader behaves as if the uncompressed data stream started with the given dictionary, which has already been read. NewReaderDict is typically used to read data compressed by NewWriterDict.

type [CorruptInputError](#)

```
type CorruptInputError int64
```

A `CorruptInputError` reports the presence of corrupt input at a given offset.

func (CorruptInputError) [Error](#)

```
func (e CorruptInputError) Error() string
```

type [InternalError](#)

```
type InternalError string
```

An InternalError reports an error in the flate code itself.

func (InternalError) [Error](#)

```
func (e InternalError) Error() string
```

type [ReadError](#)

```
type ReadError struct {  
    Offset int64 // byte offset where error occurred  
    Err     error // error returned by underlying Read  
}
```

A ReadError reports an error encountered while reading input.

func (*ReadError) [Error](#)

```
func (e *ReadError) Error() string
```

type Reader

```
type Reader interface {  
    io.Reader  
    ReadByte() (c byte, err error)  
}
```

The actual read interface needed by NewReader. If the passed in io.Reader does not also have ReadByte, the NewReader will introduce its own buffering.

type [WriteError](#)

```
type WriteError struct {  
    Offset int64 // byte offset where error occurred  
    Err     error // error returned by underlying Write  
}
```

A WriteError reports an error encountered while writing output.

func (*WriteError) [Error](#)

```
func (e *WriteError) Error() string
```

type [Writer](#)

```
type Writer struct {  
    // contains filtered or unexported fields  
}
```

A `Writer` takes data written to it and writes the compressed form of that data to an underlying writer (see `NewWriter`).

func [NewWriter](#)

```
func NewWriter(w io.Writer, level int) (*Writer, error)
```

`NewWriter` returns a new `Writer` compressing data at the given level. Following `zlib`, levels range from 1 (`BestSpeed`) to 9 (`BestCompression`); higher levels typically run slower but compress more. Level 0 (`NoCompression`) does not attempt any compression; it only adds the necessary DEFLATE framing. Level -1 (`DefaultCompression`) uses the default compression level.

If `level` is in the range `[-1, 9]` then the error returned will be `nil`. Otherwise the error returned will be non-`nil`.

func [NewWriterDict](#)

```
func NewWriterDict(w io.Writer, level int, dict []byte) (*Writer, error)
```

`NewWriterDict` is like `NewWriter` but initializes the new `Writer` with a preset dictionary. The returned `Writer` behaves as if the dictionary had been written to it without producing any compressed output. The compressed data written to `w` can only be decompressed by a `Reader` initialized with the same dictionary.

func (*Writer) [Close](#)

```
func (w *Writer) Close() error
```

`Close` flushes and closes the writer.

func (*Writer) [Flush](#)

```
func (w *Writer) Flush() error
```

Flush flushes any pending compressed data to the underlying writer. It is useful mainly in compressed network protocols, to ensure that a remote reader has enough data to reconstruct a packet. Flush does not return until the data has been written. If the underlying writer returns an error, Flush returns that error.

In the terminology of the zlib library, Flush is equivalent to Z_SYNC_FLUSH.

func (*Writer) [Write](#)

```
func (w *Writer) Write(data []byte) (n int, err error)
```

Write writes data to w, which will eventually write the compressed form of data to its underlying writer.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package gzip

```
import "compress/gzip"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package gzip implements reading and writing of gzip format compressed files, as specified in RFC 1952.

Index

[Constants](#)

[Variables](#)

[type Header](#)

[type Reader](#)

[func NewReader\(r io.Reader\) \(*Reader, error\)](#)

[func \(z *Reader\) Close\(\) error](#)

[func \(z *Reader\) Read\(p \[\]byte\) \(n int, err error\)](#)

[type Writer](#)

[func NewWriter\(w io.Writer\) *Writer](#)

[func NewWriterLevel\(w io.Writer, level int\) \(*Writer, error\)](#)

[func \(z *Writer\) Close\(\) error](#)

[func \(z *Writer\) Write\(p \[\]byte\) \(int, error\)](#)

Package files

[gunzip.go](#) [gzip.go](#)

Constants

```
const (  
    NoCompression      = flate.NoCompression  
    BestSpeed          = flate.BestSpeed  
    BestCompression    = flate.BestCompression  
    DefaultCompression = flate.DefaultCompression  
)
```

These constants are copied from the flate package, so that code that imports "compress/gzip" does not also have to import "compress/flate".

Variables

```
var (  
    // ErrChecksum is returned when reading GZIP data that has an in  
    ErrChecksum = errors.New("gzip: invalid checksum")  
    // ErrHeader is returned when reading GZIP data that has an inva  
    ErrHeader = errors.New("gzip: invalid header")  
)
```

type [Header](#)

```
type Header struct {  
    Comment string    // comment  
    Extra   []byte     // "extra data"  
    ModTime time.Time // modification time  
    Name    string     // file name  
    OS      byte       // operating system type  
}
```

The gzip file stores a header giving metadata about the compressed file. That header is exposed as the fields of the Writer and Reader structs.

type [Reader](#)

```
type Reader struct {
    Header
    // contains filtered or unexported fields
}
```

A Reader is an `io.Reader` that can be read to retrieve uncompressed data from a gzip-format compressed file.

In general, a gzip file can be a concatenation of gzip files, each with its own header. Reads from the Reader return the concatenation of the uncompressed data of each. Only the first header is recorded in the Reader fields.

Gzip files store a length and checksum of the uncompressed data. The Reader will return a `ErrChecksum` when `Read` reaches the end of the uncompressed data if it does not have the expected length or checksum. Clients should treat data returned by `Read` as tentative until they receive the `io.EOF` marking the end of the data.

func [NewReader](#)

```
func NewReader(r io.Reader) (*Reader, error)
```

`NewReader` creates a new Reader reading the given reader. The implementation buffers input and may read more data than necessary from `r`. It is the caller's responsibility to call `Close` on the Reader when done.

func (***Reader**) [Close](#)

```
func (z *Reader) Close() error
```

`Close` closes the Reader. It does not close the underlying `io.Reader`.

func (***Reader**) [Read](#)

```
func (z *Reader) Read(p []byte) (n int, err error)
```

type [Writer](#)

```
type Writer struct {
    Header
    // contains filtered or unexported fields
}
```

A `Writer` is an `io.WriteCloser` that satisfies writes by compressing data written to its wrapped `io.Writer`.

func [NewWriter](#)

```
func NewWriter(w io.Writer) *Writer
```

`NewWriter` creates a new `Writer` that satisfies writes by compressing data written to `w`.

It is the caller's responsibility to call `Close` on the `WriteCloser` when done. Writes may be buffered and not flushed until `Close`.

Callers that wish to set the fields in `Writer.Header` must do so before the first call to `Write` or `Close`. The `Comment` and `Name` header fields are UTF-8 strings in Go, but the underlying format requires NUL-terminated ISO 8859-1 (Latin-1). NUL or non-Latin-1 runes in those strings will lead to an error on `Write`.

func [NewWriterLevel](#)

```
func NewWriterLevel(w io.Writer, level int) (*Writer, error)
```

`NewWriterLevel` is like `NewWriter` but specifies the compression level instead of assuming `DefaultCompression`.

The compression level can be `DefaultCompression`, `NoCompression`, or any integer value between `BestSpeed` and `BestCompression` inclusive. The error returned will be `nil` if the level is valid.

func (*Writer) [Close](#)

```
func (z *Writer) Close() error
```

Close closes the Writer. It does not close the underlying io.Writer.

func (*Writer) [Write](#)

```
func (z *Writer) Write(p []byte) (int, error)
```

Write writes a compressed form of p to the underlying io.Writer. The compressed bytes are not necessarily flushed until the Writer is closed.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package lzw

```
import "compress/lzw"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package lzw implements the Lempel-Ziv-Welch compressed data format, described in T. A. Welch, "A Technique for High-Performance Data Compression", *Computer*, 17(6) (June 1984), pp 8-19.

In particular, it implements LZW as used by the GIF, TIFF and PDF file formats, which means variable-width codes up to 12 bits and the first two non-literal codes are a clear code and an EOF code.

Index

[func NewReader\(r io.Reader, order Order, litWidth int\) io.ReadCloser](#)
[func NewWriter\(w io.Writer, order Order, litWidth int\) io.WriteCloser](#)
[type Order](#)

Package files

[reader.go](#) [writer.go](#)

func [NewReader](#)

```
func NewReader(r io.Reader, order Order, litWidth int) io.ReadCloser
```

NewReader creates a new `io.ReadCloser` that satisfies reads by decompressing the data read from `r`. It is the caller's responsibility to call `Close` on the `ReadCloser` when finished reading. The number of bits to use for literal codes, `litWidth`, must be in the range `[2,8]` and is typically `8`.

func [NewWriter](#)

```
func NewWriter(w io.Writer, order Order, litWidth int) io.WriteClose
```

NewWriter creates a new io.WriteCloser that satisfies writes by compressing the data and writing it to w. It is the caller's responsibility to call Close on the WriteCloser when finished writing. The number of bits to use for literal codes, litWidth, must be in the range [2,8] and is typically 8.

type [Order](#)

type Order int

Order specifies the bit ordering in an LZW data stream.

```
const (  
    // LSB means Least Significant Bits first, as used in the GIF fi  
    LSB Order = iota  
    // MSB means Most Significant Bits first, as used in the TIFF an  
    // file formats.  
    MSB  
)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package zlib

```
import "compress/zlib"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `zlib` implements reading and writing of `zlib` format compressed data, as specified in RFC 1950.

The implementation provides filters that uncompress during reading and compress during writing. For example, to write compressed data to a buffer:

```
var b bytes.Buffer
w, err := zlib.NewWriter(&b)
w.Write([]byte("hello, world\n"))
w.Close()
```

and to read that data back:

```
r, err := zlib.NewReader(&b)
io.Copy(os.Stdout, r)
r.Close()
```

Index

[Constants](#)

[Variables](#)

[func NewReader\(r io.Reader\) \(io.ReadCloser, error\)](#)

[func NewReaderDict\(r io.Reader, dict \[\]byte\) \(io.ReadCloser, error\)](#)

[type Writer](#)

[func NewWriter\(w io.Writer\) *Writer](#)

[func NewWriterLevel\(w io.Writer, level int\) \(*Writer, error\)](#)

[func NewWriterLevelDict\(w io.Writer, level int, dict \[\]byte\) \(*Writer, error\)](#)

[func \(z *Writer\) Close\(\) error](#)

[func \(z *Writer\) Flush\(\) error](#)

[func \(z *Writer\) Write\(p \[\]byte\) \(n int, err error\)](#)

Package files

[reader.go](#) [writer.go](#)

Constants

```
const (  
    NoCompression      = flate.NoCompression  
    BestSpeed          = flate.BestSpeed  
    BestCompression    = flate.BestCompression  
    DefaultCompression = flate.DefaultCompression  
)
```

These constants are copied from the flate package, so that code that imports "compress/zlib" does not also have to import "compress/flate".

Variables

```
var (  
    // ErrChecksum is returned when reading ZLIB data that has an in  
    ErrChecksum = errors.New("zlib: invalid checksum")  
    // ErrDictionary is returned when reading ZLIB data that has an  
    ErrDictionary = errors.New("zlib: invalid dictionary")  
    // ErrHeader is returned when reading ZLIB data that has an inva  
    ErrHeader = errors.New("zlib: invalid header")  
)
```

func [NewReader](#)

```
func NewReader(r io.Reader) (io.ReadCloser, error)
```

NewReader creates a new `io.ReadCloser` that satisfies reads by decompressing data read from `r`. The implementation buffers input and may read more data than necessary from `r`. It is the caller's responsibility to call `Close` on the `ReadCloser` when done.

func [NewReaderDict](#)

```
func NewReaderDict(r io.Reader, dict []byte) (io.ReadCloser, error)
```

NewReaderDict is like NewReader but uses a preset dictionary. NewReaderDict ignores the dictionary if the compressed data does not refer to it.

type [Writer](#)

```
type Writer struct {  
    // contains filtered or unexported fields  
}
```

A `Writer` takes data written to it and writes the compressed form of that data to an underlying writer (see `NewWriter`).

func [NewWriter](#)

```
func NewWriter(w io.Writer) *Writer
```

`NewWriter` creates a new `Writer` that satisfies writes by compressing data written to `w`.

It is the caller's responsibility to call `Close` on the `Writer` when done. Writes may be buffered and not flushed until `Close`.

func [NewWriterLevel](#)

```
func NewWriterLevel(w io.Writer, level int) (*Writer, error)
```

`NewWriterLevel` is like `NewWriter` but specifies the compression level instead of assuming `DefaultCompression`.

The compression level can be `DefaultCompression`, `NoCompression`, or any integer value between `BestSpeed` and `BestCompression` inclusive. The error returned will be `nil` if the level is valid.

func [NewWriterLevelDict](#)

```
func NewWriterLevelDict(w io.Writer, level int, dict []byte) (*Writer, error)
```

`NewWriterLevelDict` is like `NewWriterLevel` but specifies a dictionary to compress with.

The dictionary may be `nil`. If not, its contents should not be modified until the `Writer` is closed.

func (*Writer) [Close](#)

```
func (z *Writer) Close() error
```

Calling Close does not close the wrapped io.Writer originally passed to NewWriter.

func (*Writer) [Flush](#)

```
func (z *Writer) Flush() error
```

Flush flushes the Writer to its underlying io.Writer.

func (*Writer) [Write](#)

```
func (z *Writer) Write(p []byte) (n int, err error)
```

Write writes a compressed form of p to the underlying io.Writer. The compressed bytes are not necessarily flushed until the Writer is closed or explicitly flushed.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg/container

Name	Synopsis
heap	Package heap provides heap operations for any type that implements heap.Interface.
list	Package list implements a doubly linked list.
ring	Package ring implements operations on circular lists.

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package heap

```
import "container/heap"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package heap provides heap operations for any type that implements heap.Interface. A heap is a tree with the property that each node is the highest-valued node in its subtree.

A heap is a common way to implement a priority queue. To build a priority queue, implement the Heap interface with the (negative) priority as the ordering for the Less method, so Push adds items while Pop removes the highest-priority item from the queue. The Examples include such an implementation; the file example_test.go has the complete source.

? Example

? Example

This example pushes 10 items into a PriorityQueue and takes them out in order of priority.

Code:

```
// This example demonstrates a priority queue built using the heap i
package heap_test

import (
    "container/heap"
    "fmt"
)

// An Item is something we manage in a priority queue.
type Item struct {
    value    string // The value of the item; arbitrary.
    priority int    // The priority of the item in the queue.
    // The index is needed by changePriority and is maintained by th
    index int // The index of the item in the heap.
}

// A PriorityQueue implements heap.Interface and holds Items.
type PriorityQueue []*Item

func (pq PriorityQueue) Len() int { return len(pq) }

func (pq PriorityQueue) Less(i, j int) bool {
```

```

    // We want Pop to give us the highest, not lowest, priority so w
    return pq[i].priority > pq[j].priority
}

func (pq PriorityQueue) Swap(i, j int) {
    pq[i], pq[j] = pq[j], pq[i]
    pq[i].index = i
    pq[j].index = j
}

func (pq *PriorityQueue) Push(x interface{}) {
    // Push and Pop use pointer receivers because they modify the sl
    // not just its contents.
    // To simplify indexing expressions in these methods, we save a
    // slice object. We could instead write (*pq)[i].
    a := *pq
    n := len(a)
    a = a[0 : n+1]
    item := x.(*Item)
    item.index = n
    a[n] = item
    *pq = a
}

func (pq *PriorityQueue) Pop() interface{} {
    a := *pq
    n := len(a)
    item := a[n-1]
    item.index = -1 // for safety
    *pq = a[0 : n-1]
    return item
}

// update is not used by the example but shows how to take the top i
// the queue, update its priority and value, and put it back.
func (pq *PriorityQueue) update(value string, priority int) {
    item := heap.Pop(pq).(*Item)
    item.value = value
    item.priority = priority
    heap.Push(pq, item)
}

// changePriority is not used by the example but shows how to change
// priority of an arbitrary item.
func (pq *PriorityQueue) changePriority(item *Item, priority int) {
    heap.Remove(pq, item.index)
    item.priority = priority
    heap.Push(pq, item)
}

```

```

// This example pushes 10 items into a PriorityQueue and takes them
// order of priority.
func Example() {
    const nItem = 10
    // Random priorities for the items (a permutation of 0..9, times
    priorities := [nItem]int{
        77, 22, 44, 55, 11, 88, 33, 99, 00, 66,
    }
    values := [nItem]string{
        "zero", "one", "two", "three", "four", "five", "six", "seven
    }
    // Create a priority queue and put some items in it.
    pq := make(PriorityQueue, 0, nItem)
    for i := 0; i < cap(pq); i++ {
        item := &Item{
            value:    values[i],
            priority: priorities[i],
        }
        heap.Push(&pq, item)
    }
    // Take the items out; should arrive in decreasing priority orde
    // For example, the highest priority (99) is the seventh item, s
    for i := 0; i < nItem; i++ {
        item := heap.Pop(&pq).(*Item)
        fmt.Printf("%.2d:%s ", item.priority, item.value)
    }
    // Output:
    // 99:seven 88:five 77:zero 66:nine 55:three 44:two 33:six 22:on
}

```

Index

[func Init\(h Interface\)](#)
[func Pop\(h Interface\) interface{}](#)
[func Push\(h Interface, x interface{}\)](#)
[func Remove\(h Interface, i int\) interface{}](#)
[type Interface](#)

Examples

[Package](#)

Package files

[heap.go](#)

func Init

```
func Init(h Interface)
```

A heap must be initialized before any of the heap operations can be used. Init is idempotent with respect to the heap invariants and may be called whenever the heap invariants may have been invalidated. Its complexity is $O(n)$ where $n = h.Len()$.

func Pop

```
func Pop(h Interface) interface{}
```

Pop removes the minimum element (according to Less) from the heap and returns it. The complexity is $O(\log(n))$ where $n = h.Len()$. Same as `Remove(h, 0)`.

func Push

```
func Push(h Interface, x interface{})
```

Push pushes the element x onto the heap. The complexity is $O(\log(n))$ where $n = h.Len()$.

func Remove

```
func Remove(h Interface, i int) interface{}
```

Remove removes the element at index i from the heap. The complexity is $O(\log(n))$ where $n = h.Len()$.

type [Interface](#)

```
type Interface interface {
    sort.Interface
    Push(x interface{}) // add x as element Len()
    Pop() interface{}   // remove and return element Len() - 1.
}
```

Any type that implements `heap.Interface` may be used as a min-heap with the following invariants (established after `Init` has been called or if the data is empty or sorted):

`!h.Less(j, i)` for $0 \leq i < h.Len()$ and $j = 2*i+1$ or $2*i+2$ and $j < h.Len()$.

Note that `Push` and `Pop` in this interface are for package `heap`'s implementation to call. To add and remove things from the heap, use `heap.Push` and `heap.Pop`.

Build version `go1.0.1`.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package list

```
import "container/list"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package list implements a doubly linked list.

To iterate over a list (where l is a *List):

```
for e := l.Front(); e != nil; e = e.Next() {  
    // do something with e.Value  
}
```

Index

type Element

func (e *Element) Next() *Element

func (e *Element) Prev() *Element

type List

func New() *List

func (l *List) Back() *Element

func (l *List) Front() *Element

func (l *List) Init() *List

func (l *List) InsertAfter(value interface{}, mark *Element) *Element

func (l *List) InsertBefore(value interface{}, mark *Element) *Element

func (l *List) Len() int

func (l *List) MoveToBack(e *Element)

func (l *List) MoveToFront(e *Element)

func (l *List) PushBack(value interface{}) *Element

func (l *List) PushBackList(ol *List)

func (l *List) PushFront(value interface{}) *Element

func (l *List) PushFrontList(ol *List)

func (l *List) Remove(e *Element) interface{}

Package files

[list.go](#)

type [Element](#)

```
type Element struct {  
    // The contents of this list element.  
    Value interface{}  
    // contains filtered or unexported fields  
}
```

Element is an element in the linked list.

func (*Element) [Next](#)

```
func (e *Element) Next() *Element
```

Next returns the next list element or nil.

func (*Element) [Prev](#)

```
func (e *Element) Prev() *Element
```

Prev returns the previous list element or nil.

type [List](#)

```
type List struct {  
    // contains filtered or unexported fields  
}
```

List represents a doubly linked list. The zero value for List is an empty list ready to use.

func [New](#)

```
func New() *List
```

New returns an initialized list.

func ([*List](#)) [Back](#)

```
func (l *List) Back() *Element
```

Back returns the last element in the list.

func ([*List](#)) [Front](#)

```
func (l *List) Front() *Element
```

Front returns the first element in the list.

func ([*List](#)) [Init](#)

```
func (l *List) Init() *List
```

Init initializes or clears a List.

func ([*List](#)) [InsertAfter](#)

```
func (l *List) InsertAfter(value interface{}, mark *Element) *Element
```

InsertAfter inserts the value immediately after mark and returns a new Element containing the value.

func (*List) [InsertBefore](#)

```
func (l *List) InsertBefore(value interface{}, mark *Element) *Element
```

InsertBefore inserts the value immediately before mark and returns a new Element containing the value.

func (*List) [Len](#)

```
func (l *List) Len() int
```

Len returns the number of elements in the list.

func (*List) [MoveToBack](#)

```
func (l *List) MoveToBack(e *Element)
```

MoveToBack moves the element to the back of the list.

func (*List) [MoveToFront](#)

```
func (l *List) MoveToFront(e *Element)
```

MoveToFront moves the element to the front of the list.

func (*List) [PushBack](#)

```
func (l *List) PushBack(value interface{}) *Element
```

PushBack inserts the value at the back of the list and returns a new Element containing the value.

func (*List) [PushBackList](#)

```
func (l *List) PushBackList(ol *List)
```

PushBackList inserts each element of ol at the back of the list.

func (*List) [PushFront](#)

```
func (l *List) PushFront(value interface{}) *Element
```

PushFront inserts the value at the front of the list and returns a new Element containing the value.

func (*List) [PushFrontList](#)

```
func (l *List) PushFrontList(ol *List)
```

PushFrontList inserts each element of ol at the front of the list. The ordering of the passed list is preserved.

func (*List) [Remove](#)

```
func (l *List) Remove(e *Element) interface{}
```

Remove removes the element from the list and returns its Value.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package ring

```
import "container/ring"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package ring implements operations on circular lists.

Index

[type Ring](#)

[func New\(n int\) *Ring](#)

[func \(r *Ring\) Do\(f func\(interface{}\)\)](#)

[func \(r *Ring\) Len\(\) int](#)

[func \(r *Ring\) Link\(s *Ring\) *Ring](#)

[func \(r *Ring\) Move\(n int\) *Ring](#)

[func \(r *Ring\) Next\(\) *Ring](#)

[func \(r *Ring\) Prev\(\) *Ring](#)

[func \(r *Ring\) Unlink\(n int\) *Ring](#)

Package files

[ring.go](#)

type [Ring](#)

```
type Ring struct {
    Value interface{} // for use by client; untouched by this library
    // contains filtered or unexported fields
}
```

A Ring is an element of a circular list, or ring. Rings do not have a beginning or end; a pointer to any ring element serves as reference to the entire ring. Empty rings are represented as nil Ring pointers. The zero value for a Ring is a one-element ring with a nil Value.

func [New](#)

```
func New(n int) *Ring
```

New creates a ring of n elements.

func (*Ring) [Do](#)

```
func (r *Ring) Do(f func(interface{}))
```

Do calls function f on each element of the ring, in forward order. The behavior of Do is undefined if f changes *r.

func (*Ring) [Len](#)

```
func (r *Ring) Len() int
```

Len computes the number of elements in ring r. It executes in time proportional to the number of elements.

func (*Ring) [Link](#)

```
func (r *Ring) Link(s *Ring) *Ring
```

Link connects ring r with ring s such that r.Next() becomes s and returns the original value for r.Next(). r must not be empty.

If r and s point to the same ring, linking them removes the elements between r and s from the ring. The removed elements form a subring and the result is a reference to that subring (if no elements were removed, the result is still the original value for r.Next(), and not nil).

If r and s point to different rings, linking them creates a single ring with the elements of s inserted after r. The result points to the element following the last element of s after insertion.

func (*Ring) [Move](#)

```
func (r *Ring) Move(n int) *Ring
```

Move moves $n \% r.Len()$ elements backward ($n < 0$) or forward ($n \geq 0$) in the ring and returns that ring element. r must not be empty.

func (*Ring) [Next](#)

```
func (r *Ring) Next() *Ring
```

Next returns the next ring element. r must not be empty.

func (*Ring) [Prev](#)

```
func (r *Ring) Prev() *Ring
```

Prev returns the previous ring element. r must not be empty.

func (*Ring) [Unlink](#)

```
func (r *Ring) Unlink(n int) *Ring
```

Unlink removes $n \% r.Len()$ elements from the ring r, starting at r.Next(). If $n \% r.Len() == 0$, r remains unchanged. The result is the removed subring. r must not be empty.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package crypto

```
import "crypto"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package `crypto` collects common cryptographic constants.

Index

[func RegisterHash\(h Hash, f func\(\) hash.Hash\)](#)

[type Hash](#)

[func \(h Hash\) Available\(\) bool](#)

[func \(h Hash\) New\(\) hash.Hash](#)

[func \(h Hash\) Size\(\) int](#)

[type PrivateKey](#)

Package files

[crypto.go](#)

func [RegisterHash](#)

```
func RegisterHash(h Hash, f func() hash.Hash)
```

RegisterHash registers a function that returns a new instance of the given hash function. This is intended to be called from the init function in packages that implement hash functions.

type [Hash](#)

```
type Hash uint
```

Hash identifies a cryptographic hash function that is implemented in another package.

```
const (
    MD4          Hash = 1 + iota // import code.google.com/p/go.crypto/
    MD5          // import crypto/md5
    SHA1         // import crypto/sha1
    SHA224       // import crypto/sha256
    SHA256       // import crypto/sha256
    SHA384       // import crypto/sha512
    SHA512       // import crypto/sha512
    MD5SHA1      // no implementation; MD5+SHA1 used fo
    RIPEMD160    // import code.google.com/p/go.crypto/
)
)
```

func (Hash) [Available](#)

```
func (h Hash) Available() bool
```

Available reports whether the given hash function is linked into the binary.

func (Hash) [New](#)

```
func (h Hash) New() hash.Hash
```

New returns a new hash.Hash calculating the given hash function. New panics if the hash function is not linked into the binary.

func (Hash) [Size](#)

```
func (h Hash) Size() int
```

Size returns the length, in bytes, of a digest resulting from the given hash function. It doesn't require that the hash function in question be linked into the program.

type PrivateKey

```
type PrivateKey interface{}
```

PrivateKey represents a private key using an unspecified algorithm.

Subdirectories

Name	Synopsis
aes	Package aes implements AES encryption (formerly Rijndael), as defined in U.S. Federal Information Processing Standards Publication 197.
cipher	Package cipher implements standard block cipher modes that can be wrapped around low-level block cipher implementations.
des	Package des implements the Data Encryption Standard (DES) and the Triple Data Encryption Algorithm (TDEA) as defined in U.S. Federal Information Processing Standards Publication 46-3.
dsa	Package dsa implements the Digital Signature Algorithm, as defined in FIPS 186-3.
ecdsa	Package ecdsa implements the Elliptic Curve Digital Signature Algorithm, as defined in FIPS 186-3.
elliptic	Package elliptic implements several standard elliptic curves over prime fields.
hmac	Package hmac implements the Keyed-Hash Message Authentication Code (HMAC) as defined in U.S. Federal Information Processing Standards Publication 198.
md5	Package md5 implements the MD5 hash algorithm as defined in RFC 1321.
rand	Package rand implements a cryptographically secure pseudorandom number generator.
rc4	Package rc4 implements RC4 encryption, as defined in Bruce Schneier's Applied Cryptography.
rsa	Package rsa implements RSA encryption as specified in PKCS#1.
sha1	Package sha1 implements the SHA1 hash algorithm as defined in RFC 3174.
sha256	Package sha256 implements the SHA224 and SHA256 hash algorithms as defined in FIPS 180-2.
sha512	Package sha512 implements the SHA384 and SHA512 hash algorithms as defined in FIPS 180-2.
	Package subtle implements functions that are often useful in

[subtle](#) cryptographic code but require careful thought to use correctly.

[tls](#) Package tls partially implements TLS 1.0, as specified in RFC 2246.

[x509](#) Package x509 parses X.509-encoded keys and certificates.

[pkix](#) Package pkix contains shared, low level structures used for ASN.1 parsing and serialization of X.509 certificates, CRL and OCSP.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package aes

```
import "crypto/aes"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package aes implements AES encryption (formerly Rijndael), as defined in U.S. Federal Information Processing Standards Publication 197.

Index

Constants

[func NewCipher\(key \[\]byte\) \(cipher.Block, error\)](#)

[type KeySizeError](#)

[func \(k KeySizeError\) Error\(\) string](#)

Package files

[block.go](#) [cipher.go](#) [const.go](#)

Constants

```
const BlockSize = 16
```

The AES block size in bytes.

func [NewCipher](#)

```
func NewCipher(key []byte) (cipher.Block, error)
```

NewCipher creates and returns a new cipher.Block. The key argument should be the AES key, either 16, 24, or 32 bytes to select AES-128, AES-192, or AES-256.

type [KeySizeError](#)

```
type KeySizeError int
```

func (KeySizeError) [Error](#)

```
func (k KeySizeError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package cipher

```
import "crypto/cipher"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package cipher implements standard block cipher modes that can be wrapped around low-level block cipher implementations. See http://csrc.nist.gov/groups/ST/toolkit/BCM/current_modes.html and NIST Special Publication 800-38A.

Index

[type Block](#)

[type BlockMode](#)

[func NewCBCDecrypter\(b Block, iv \[\]byte\) BlockMode](#)

[func NewCBCEncrypter\(b Block, iv \[\]byte\) BlockMode](#)

[type Stream](#)

[func NewCFBDecrypter\(block Block, iv \[\]byte\) Stream](#)

[func NewCFBEncrypter\(block Block, iv \[\]byte\) Stream](#)

[func NewCTR\(block Block, iv \[\]byte\) Stream](#)

[func NewOFB\(b Block, iv \[\]byte\) Stream](#)

[type StreamReader](#)

[func \(r StreamReader\) Read\(dst \[\]byte\) \(n int, err error\)](#)

[type StreamWriter](#)

[func \(w StreamWriter\) Close\(\) error](#)

[func \(w StreamWriter\) Write\(src \[\]byte\) \(n int, err error\)](#)

Package files

[cbc.go](#) [cfb.go](#) [cipher.go](#) [ctr.go](#) [io.go](#) [ofb.go](#)

type [Block](#)

```
type Block interface {
    // BlockSize returns the cipher's block size.
    BlockSize() int

    // Encrypt encrypts the first block in src into dst.
    // Dst and src may point at the same memory.
    Encrypt(dst, src []byte)

    // Decrypt decrypts the first block in src into dst.
    // Dst and src may point at the same memory.
    Decrypt(dst, src []byte)
}
```

A Block represents an implementation of block cipher using a given key. It provides the capability to encrypt or decrypt individual blocks. The mode implementations extend that capability to streams of blocks.

type [BlockMode](#)

```
type BlockMode interface {
    // BlockSize returns the mode's block size.
    BlockSize() int

    // CryptBlocks encrypts or decrypts a number of blocks. The leng
    // src must be a multiple of the block size. Dst and src may poi
    // the same memory.
    CryptBlocks(dst, src []byte)
}
```

A BlockMode represents a block cipher running in a block-based mode (CBC, ECB etc).

func [NewCBCDecrypter](#)

```
func NewCBCDecrypter(b Block, iv []byte) BlockMode
```

NewCBCDecrypter returns a BlockMode which decrypts in cipher block chaining mode, using the given Block. The length of iv must be the same as the Block's block size and must match the iv used to encrypt the data.

func [NewCBCEncrypter](#)

```
func NewCBCEncrypter(b Block, iv []byte) BlockMode
```

NewCBCEncrypter returns a BlockMode which encrypts in cipher block chaining mode, using the given Block. The length of iv must be the same as the Block's block size.

type [Stream](#)

```
type Stream interface {  
    // XORKeyStream XORs each byte in the given slice with a byte fr  
    // cipher's key stream. Dst and src may point to the same memory  
    XORKeyStream(dst, src []byte)  
}
```

A Stream represents a stream cipher.

func [NewCFBDecrypter](#)

```
func NewCFBDecrypter(block Block, iv []byte) Stream
```

NewCFBDecrypter returns a Stream which decrypts with cipher feedback mode, using the given Block. The iv must be the same length as the Block's block size.

func [NewCFBEncrypter](#)

```
func NewCFBEncrypter(block Block, iv []byte) Stream
```

NewCFBEncrypter returns a Stream which encrypts with cipher feedback mode, using the given Block. The iv must be the same length as the Block's block size.

func [NewCTR](#)

```
func NewCTR(block Block, iv []byte) Stream
```

NewCTR returns a Stream which encrypts/decrypts using the given Block in counter mode. The length of iv must be the same as the Block's block size.

func [NewOFB](#)

```
func NewOFB(b Block, iv []byte) Stream
```

NewOFB returns a Stream that encrypts or decrypts using the block cipher b in output feedback mode. The initialization vector iv's length must be equal to b's block size.

type [StreamReader](#)

```
type StreamReader struct {  
    S Stream  
    R io.Reader  
}
```

StreamReader wraps a Stream into an io.Reader. It calls XORKeyStream to process each slice of data which passes through.

func (StreamReader) [Read](#)

```
func (r StreamReader) Read(dst []byte) (n int, err error)
```

type [StreamWriter](#)

```
type StreamWriter struct {  
    S    Stream  
    W    io.Writer  
    Err  error  
}
```

StreamWriter wraps a Stream into an io.Writer. It calls XORKeyStream to process each slice of data which passes through. If any Write call returns short then the StreamWriter is out of sync and must be discarded.

func (StreamWriter) [Close](#)

```
func (w StreamWriter) Close() error
```

func (StreamWriter) [Write](#)

```
func (w StreamWriter) Write(src []byte) (n int, err error)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package des

```
import "crypto/des"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package des implements the Data Encryption Standard (DES) and the Triple Data Encryption Algorithm (TDEA) as defined in U.S. Federal Information Processing Standards Publication 46-3.

Index

Constants

[func NewCipher\(key \[\]byte\) \(cipher.Block, error\)](#)

[func NewTripleDESCipher\(key \[\]byte\) \(cipher.Block, error\)](#)

[type KeySizeError](#)

[func \(k KeySizeError\) Error\(\) string](#)

Package files

[block.go](#) [cipher.go](#) [const.go](#)

Constants

```
const BlockSize = 8
```

The DES block size in bytes.

func [NewCipher](#)

```
func NewCipher(key []byte) (cipher.Block, error)
```

NewCipher creates and returns a new cipher.Block.

func [NewTripleDESCipher](#)

```
func NewTripleDESCipher(key []byte) (cipher.Block, error)
```

NewTripleDESCipher creates and returns a new cipher.Block.

type [KeySizeError](#)

```
type KeySizeError int
```

func (KeySizeError) Error

```
func (k KeySizeError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package dsa

```
import "crypto/dsa"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package dsa implements the Digital Signature Algorithm, as defined in FIPS 186-3.

Index

Variables

func GenerateKey(priv *PrivateKey, rand io.Reader) error

func GenerateParameters(params *Parameters, rand io.Reader, sizes
ParameterSizes) (err error)

func Sign(rand io.Reader, priv *PrivateKey, hash []byte) (r, s *big.Int, err
error)

func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool

type ParameterSizes

type Parameters

type PrivateKey

type PublicKey

Package files

dsa.go

Variables

```
var ErrInvalidPublicKey = errors.New("crypto/dsa: invalid public key")
```

`ErrInvalidPublicKey` results when a public key is not usable by this code. FIPS is quite strict about the format of DSA keys, but other code may be less so. Thus, when using keys which may have been generated by other code, this error must be handled.

func [GenerateKey](#)

```
func GenerateKey(priv *PrivateKey, rand io.Reader) error
```

GenerateKey generates a public&private key pair. The Parameters of the PrivateKey must already be valid (see GenerateParameters).

func GenerateParameters

```
func GenerateParameters(params *Parameters, rand io.Reader, sizes Pa
```

GenerateParameters puts a random, valid set of DSA parameters into params.
This function takes many seconds, even on fast machines.

func [Sign](#)

```
func Sign(rand io.Reader, priv *PrivateKey, hash []byte) (r, s *big.
```

Sign signs an arbitrary length hash (which should be the result of hashing a larger message) using the private key, priv. It returns the signature as a pair of integers. The security of the private key depends on the entropy of rand.

Note that FIPS 186-3 section 4.6 specifies that the hash should be truncated to the byte-length of the subgroup. This function does not perform that truncation itself.

func [Verify](#)

```
func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool
```

Verify verifies the signature in r, s of hash using the public key, pub. It reports whether the signature is valid.

Note that FIPS 186-3 section 4.6 specifies that the hash should be truncated to the byte-length of the subgroup. This function does not perform that truncation itself.

type [ParameterSizes](#)

```
type ParameterSizes int
```

ParameterSizes is an enumeration of the acceptable bit lengths of the primes in a set of DSA parameters. See FIPS 186-3, section 4.2.

```
const (  
    L1024N160 ParameterSizes = iota  
    L2048N224  
    L2048N256  
    L3072N256  
)
```

type Parameters

```
type Parameters struct {  
    P, Q, G *big.Int  
}
```

Parameters represents the domain parameters for a key. These parameters can be shared across many keys. The bit length of Q must be a multiple of 8.

type [PrivateKey](#)

```
type PrivateKey struct {  
    PublicKey  
    X *big.Int  
}
```

PrivateKey represents a DSA private key.

type [PublicKey](#)

```
type PublicKey struct {  
    Parameters  
    Y *big.Int  
}
```

PublicKey represents a DSA public key.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package ecdsa

```
import "crypto/ecdsa"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `ecdsa` implements the Elliptic Curve Digital Signature Algorithm, as defined in FIPS 186-3.

Index

[func Sign\(rand io.Reader, priv *PrivateKey, hash \[\]byte\) \(r, s *big.Int, error\)](#)

[func Verify\(pub *PublicKey, hash \[\]byte, r, s *big.Int\) bool](#)

[type PrivateKey](#)

[func GenerateKey\(c elliptic.Curve, rand io.Reader\) \(priv *PrivateKey, error\)](#)

[type PublicKey](#)

Package files

ecdsa.go

func Sign

```
func Sign(rand io.Reader, priv *PrivateKey, hash []byte) (r, s *big.
```

Sign signs an arbitrary length hash (which should be the result of hashing a larger message) using the private key, priv. It returns the signature as a pair of integers. The security of the private key depends on the entropy of rand.

func [Verify](#)

```
func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool
```

Verify verifies the signature in r, s of hash using the public key, pub. It returns true iff the signature is valid.

type [PrivateKey](#)

```
type PrivateKey struct {  
    PublicKey  
    D *big.Int  
}
```

PrivateKey represents a ECDSA private key.

func [GenerateKey](#)

```
func GenerateKey(c elliptic.Curve, rand io.Reader) (priv *PrivateKey
```

GenerateKey generates a public&private key pair.

type [PublicKey](#)

```
type PublicKey struct {  
    elliptic.Curve  
    X, Y *big.Int  
}
```

PublicKey represents an ECDSA public key.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package elliptic

```
import "crypto/elliptic"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package elliptic implements several standard elliptic curves over prime fields.

Index

[func GenerateKey\(curve Curve, rand io.Reader\) \(priv \[\]byte, x, y *big.Int, err error\)](#)

[func Marshal\(curve Curve, x, y *big.Int\) \[\]byte](#)

[func Unmarshal\(curve Curve, data \[\]byte\) \(x, y *big.Int\)](#)

[type Curve](#)

[func P224\(\) Curve](#)

[func P256\(\) Curve](#)

[func P384\(\) Curve](#)

[func P521\(\) Curve](#)

[type CurveParams](#)

[func \(curve *CurveParams\) Add\(x1, y1, x2, y2 *big.Int\) \(*big.Int, *big.Int\)](#)

[func \(curve *CurveParams\) Double\(x1, y1 *big.Int\) \(*big.Int, *big.Int\)](#)

[func \(curve *CurveParams\) IsOnCurve\(x, y *big.Int\) bool](#)

[func \(curve *CurveParams\) Params\(\) *CurveParams](#)

[func \(curve *CurveParams\) ScalarBaseMult\(k \[\]byte\) \(*big.Int, *big.Int\)](#)

[func \(curve *CurveParams\) ScalarMult\(Bx, By *big.Int, k \[\]byte\) \(*big.Int, *big.Int\)](#)

Package files

[elliptic.go](#) [p224.go](#)

func [GenerateKey](#)

```
func GenerateKey(curve Curve, rand io.Reader) (priv []byte, x, y *bi
```

GenerateKey returns a public/private key pair. The private key is generated using the given reader, which must return random data.

func [Marshal](#)

```
func Marshal(curve Curve, x, y *big.Int) []byte
```

Marshal converts a point into the form specified in section 4.3.6 of ANSI X9.62.

func [Unmarshal](#)

```
func Unmarshal(curve Curve, data []byte) (x, y *big.Int)
```

Unmarshal converts a point, serialized by Marshal, into an x, y pair. On error, x = nil.

type [Curve](#)

```
type Curve interface {
    // Params returns the parameters for the curve.
    Params() *CurveParams
    // IsOnCurve returns true if the given (x,y) lies on the curve.
    IsOnCurve(x, y *big.Int) bool
    // Add returns the sum of (x1,y1) and (x2,y2)
    Add(x1, y1, x2, y2 *big.Int) (x, y *big.Int)
    // Double returns 2*(x,y)
    Double(x1, y1 *big.Int) (x, y *big.Int)
    // ScalarMult returns k*(Bx,By) where k is a number in big-endian
    ScalarMult(x1, y1 *big.Int, scalar []byte) (x, y *big.Int)
    // ScalarBaseMult returns k*G, where G is the base point of the
    // is an integer in big-endian form.
    ScalarBaseMult(scalar []byte) (x, y *big.Int)
}
```

A Curve represents a short-form Weierstrass curve with $a=-3$. See <http://www.hyperelliptic.org/EFD/g1p/auto-shortw.html>

func [P224](#)

```
func P224() Curve
```

P224 returns a Curve which implements P-224 (see FIPS 186-3, section D.2.2)

func [P256](#)

```
func P256() Curve
```

P256 returns a Curve which implements P-256 (see FIPS 186-3, section D.2.3)

func [P384](#)

```
func P384() Curve
```

P384 returns a Curve which implements P-384 (see FIPS 186-3, section D.2.4)

func [P521](#)

func P521() Curve

P256 returns a Curve which implements P-521 (see FIPS 186-3, section D.2.5)

type [CurveParams](#)

```
type CurveParams struct {
    P      *big.Int // the order of the underlying field
    N      *big.Int // the order of the base point
    B      *big.Int // the constant of the curve equation
    Gx, Gy *big.Int // (x,y) of the base point
    BitSize int    // the size of the underlying field
}
```

CurveParams contains the parameters of an elliptic curve and also provides a generic, non-constant time implementation of Curve.

func (*CurveParams) [Add](#)

```
func (curve *CurveParams) Add(x1, y1, x2, y2 *big.Int) (*big.Int, *big.Int)
```

func (*CurveParams) [Double](#)

```
func (curve *CurveParams) Double(x1, y1 *big.Int) (*big.Int, *big.Int)
```

func (*CurveParams) [IsOnCurve](#)

```
func (curve *CurveParams) IsOnCurve(x, y *big.Int) bool
```

func (*CurveParams) [Params](#)

```
func (curve *CurveParams) Params() *CurveParams
```

func (*CurveParams) [ScalarBaseMult](#)

```
func (curve *CurveParams) ScalarBaseMult(k []byte) (*big.Int, *big.Int)
```

func (*CurveParams) [ScalarMult](#)

```
func (curve *CurveParams) ScalarMult(Bx, By *big.Int, k []byte) (*big.Int, *big.Int)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package hmac

```
import "crypto/hmac"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `hmac` implements the Keyed-Hash Message Authentication Code (HMAC) as defined in U.S. Federal Information Processing Standards Publication 198. An HMAC is a cryptographic hash that uses a key to sign a message. The receiver verifies the hash by recomputing it using the same key.

Index

[func New\(h func\(\) hash.Hash, key \[\]byte\) hash.Hash](#)

Package files

[hmac.go](#)

func [New](#)

```
func New(h func() hash.Hash, key []byte) hash.Hash
```

New returns a new HMAC hash using the given hash.Hash type and key.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package md5

```
import "crypto/md5"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package md5 implements the MD5 hash algorithm as defined in RFC 1321.

Index

[Constants](#)

[func New\(\) hash.Hash](#)

Examples

[New](#)

Package files

[md5.go](#) [md5block.go](#)

Constants

```
const BlockSize = 64
```

The blocksize of MD5 in bytes.

```
const Size = 16
```

The size of an MD5 checksum in bytes.

func [New](#)

```
func New() hash.Hash
```

New returns a new hash.Hash computing the MD5 checksum.

? Example

? Example

Code:

```
h := md5.New()  
io.WriteString(h, "The fog is getting thicker!")  
io.WriteString(h, "And Leon's getting laaarger!")  
fmt.Printf("%x", h.Sum(nil))
```

Output:

```
e2c569be17396eca2a2e3c11578123ed
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package rand

```
import "crypto/rand"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package rand implements a cryptographically secure pseudorandom number generator.

Index

Variables

[func Int\(rand io.Reader, max *big.Int\) \(n *big.Int, err error\)](#)

[func Prime\(rand io.Reader, bits int\) \(p *big.Int, err error\)](#)

[func Read\(b \[\]byte\) \(n int, err error\)](#)

Package files

[rand.go](#) [rand_unix.go](#) [util.go](#)

Variables

```
var Reader io.Reader
```

Reader is a global, shared instance of a cryptographically strong pseudo-random generator. On Unix-like systems, Reader reads from `/dev/urandom`. On Windows systems, Reader uses the CryptGenRandom API.

func [Int](#)

```
func Int(rand io.Reader, max *big.Int) (*big.Int, error)
```

Int returns a uniform random value in [0, max).

func Prime

```
func Prime(rand io.Reader, bits int) (p *big.Int, err error)
```

Prime returns a number, p, of the given size, such that p is prime with high probability.

func [Read](#)

```
func Read(b []byte) (n int, err error)
```

Read is a helper function that calls Reader.Read.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package rc4

```
import "crypto/rc4"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package rc4 implements RC4 encryption, as defined in Bruce Schneier's Applied Cryptography.

Index

[type Cipher](#)

[func NewCipher\(key \[\]byte\) \(*Cipher, error\)](#)

[func \(c *Cipher\) Reset\(\)](#)

[func \(c *Cipher\) XORKeyStream\(dst, src \[\]byte\)](#)

[type KeySizeError](#)

[func \(k KeySizeError\) Error\(\) string](#)

[Bugs](#)

Package files

[rc4.go](#)

type [Cipher](#)

```
type Cipher struct {  
    // contains filtered or unexported fields  
}
```

A Cipher is an instance of RC4 using a particular key.

func [NewCipher](#)

```
func NewCipher(key []byte) (*Cipher, error)
```

NewCipher creates and returns a new Cipher. The key argument should be the RC4 key, at least 1 byte and at most 256 bytes.

func (***Cipher**) [Reset](#)

```
func (c *Cipher) Reset()
```

Reset zeros the key data so that it will no longer appear in the process's memory.

func (***Cipher**) [XORKeyStream](#)

```
func (c *Cipher) XORKeyStream(dst, src []byte)
```

XORKeyStream sets dst to the result of XORing src with the key stream. Dst and src may be the same slice but otherwise should not overlap.

type [KeySizeError](#)

```
type KeySizeError int
```

func (KeySizeError) [Error](#)

```
func (k KeySizeError) Error() string
```

Bugs

RC4 is in common use but has design weaknesses that make it a poor choice for new protocols.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package rsa

```
import "crypto/rsa"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package rsa implements RSA encryption as specified in PKCS#1.

Index

Variables

[func DecryptOAEP\(hash hash.Hash, random io.Reader, priv *PrivateKey, ciphertext \[\]byte, label \[\]byte\) \(msg \[\]byte, err error\)](#)

[func DecryptPKCS1v15\(rand io.Reader, priv *PrivateKey, ciphertext \[\]byte\) \(out \[\]byte, err error\)](#)

[func DecryptPKCS1v15SessionKey\(rand io.Reader, priv *PrivateKey, ciphertext \[\]byte, key \[\]byte\) \(err error\)](#)

[func EncryptOAEP\(hash hash.Hash, random io.Reader, pub *PublicKey, msg \[\]byte, label \[\]byte\) \(out \[\]byte, err error\)](#)

[func EncryptPKCS1v15\(rand io.Reader, pub *PublicKey, msg \[\]byte\) \(out \[\]byte, err error\)](#)

[func SignPKCS1v15\(rand io.Reader, priv *PrivateKey, hash crypto.Hash, hashed \[\]byte\) \(s \[\]byte, err error\)](#)

[func VerifyPKCS1v15\(pub *PublicKey, hash crypto.Hash, hashed \[\]byte, sig \[\]byte\) \(err error\)](#)

[type CRTValue](#)

[type PrecomputedValues](#)

[type PrivateKey](#)

[func GenerateKey\(random io.Reader, bits int\) \(priv *PrivateKey, err error\)](#)

[func GenerateMultiPrimeKey\(random io.Reader, nprimes int, bits int\) \(priv *PrivateKey, err error\)](#)

[func \(priv *PrivateKey\) Precompute\(\)](#)

[func \(priv *PrivateKey\) Validate\(\) error](#)

[type PublicKey](#)

Package files

[pkcs1v15.go](#) [rsa.go](#)

Variables

```
var ErrDecryption = errors.New("crypto/rsa: decryption error")
```

ErrDecryption represents a failure to decrypt a message. It is deliberately vague to avoid adaptive attacks.

```
var ErrMessageTooLong = errors.New("crypto/rsa: message too long for
```

ErrMessageTooLong is returned when attempting to encrypt a message which is too large for the size of the public key.

```
var ErrVerification = errors.New("crypto/rsa: verification error")
```

ErrVerification represents a failure to verify a signature. It is deliberately vague to avoid adaptive attacks.

func DecryptOAEP

```
func DecryptOAEP(hash hash.Hash, random io.Reader, priv *PrivateKey,
```

DecryptOAEP decrypts ciphertext using RSA-OAEP. If random != nil, DecryptOAEP uses RSA blinding to avoid timing side-channel attacks.

func [DecryptPKCS1v15](#)

```
func DecryptPKCS1v15(rand io.Reader, priv *PrivateKey, ciphertext []
```

DecryptPKCS1v15 decrypts a plaintext using RSA and the padding scheme from PKCS#1 v1.5. If rand != nil, it uses RSA blinding to avoid timing side-channel attacks.

func [DecryptPKCS1v15SessionKey](#)

```
func DecryptPKCS1v15SessionKey(rand io.Reader, priv *PrivateKey, cip
```

DecryptPKCS1v15SessionKey decrypts a session key using RSA and the padding scheme from PKCS#1 v1.5. If rand != nil, it uses RSA blinding to avoid timing side-channel attacks. It returns an error if the ciphertext is the wrong length or if the ciphertext is greater than the public modulus. Otherwise, no error is returned. If the padding is valid, the resulting plaintext message is copied into key. Otherwise, key is unchanged. These alternatives occur in constant time. It is intended that the user of this function generate a random session key beforehand and continue the protocol with the resulting value. This will remove any possibility that an attacker can learn any information about the plaintext. See “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1”, Daniel Bleichenbacher, Advances in Cryptology (Crypto '98).

func [EncryptOAEP](#)

```
func EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey, m
```

EncryptOAEP encrypts the given message with RSA-OAEP. The message must be no longer than the length of the public modulus less twice the hash length plus 2.

func [EncryptPKCS1v15](#)

```
func EncryptPKCS1v15(rand io.Reader, pub *PublicKey, msg []byte) (ou
```

EncryptPKCS1v15 encrypts the given message with RSA and the padding scheme from PKCS#1 v1.5. The message must be no longer than the length of the public modulus minus 11 bytes. **WARNING:** use of this function to encrypt plaintexts other than session keys is dangerous. Use RSA OAEP in new protocols.

func [SignPKCS1v15](#)

```
func SignPKCS1v15(rand io.Reader, priv *PrivateKey, hash crypto.Hash
```

SignPKCS1v15 calculates the signature of hashed using RSASSA-PKCS1-V1_5-SIGN from RSA PKCS#1 v1.5. Note that hashed must be the result of hashing the input message using the given hash function.

func [VerifyPKCS1v15](#)

```
func VerifyPKCS1v15(pub *PublicKey, hash crypto.Hash, hashed []byte,
```

VerifyPKCS1v15 verifies an RSA PKCS#1 v1.5 signature. hashed is the result of hashing the input message using the given hash function and sig is the signature. A valid signature is indicated by returning a nil error.

type [CRTValue](#)

```
type CRTValue struct {  
    Exp    *big.Int // D mod (prime-1).  
    Coeff  *big.Int // RCoeff 1 mod Prime.  
    R      *big.Int // product of primes prior to this (inc p and q).  
}
```

CRTValue contains the precomputed chinese remainder theorem values.

type PrecomputedValues

```
type PrecomputedValues struct {
    Dp, Dq *big.Int // D mod (P-1) (or mod Q-1)
    Qinv  *big.Int

    // CRTValues is used for the 3rd and subsequent primes. Due to a
    // historical accident, the CRT for the first two primes is hand
    // differently in PKCS#1 and interoperability is sufficiently
    // important that we mirror this.
    CRTValues []CRTValue
}
```

type [PrivateKey](#)

```
type PrivateKey struct {
    PublicKey          // public part.
    D                 *big.Int // private exponent
    Primes            []*big.Int

    // Precomputed contains precomputed values that speed up private
    // operations, if available.
    Precomputed PrecomputedValues
}
```

A PrivateKey represents an RSA key

func [GenerateKey](#)

```
func GenerateKey(random io.Reader, bits int) (priv *PrivateKey, err
```

GenerateKey generates an RSA keypair of the given bit size.

func [GenerateMultiPrimeKey](#)

```
func GenerateMultiPrimeKey(random io.Reader, nprimes int, bits int)
```

GenerateMultiPrimeKey generates a multi-prime RSA keypair of the given bit size, as suggested in [1]. Although the public keys are compatible (actually, indistinguishable) from the 2-prime case, the private keys are not. Thus it may not be possible to export multi-prime private keys in certain formats or to subsequently import them into other code.

Table 1 in [2] suggests maximum numbers of primes for a given size.

[1] US patent 4405829 (1972, expired) [2]

<http://www.cacr.math.uwaterloo.ca/techreports/2006/cacr2006-16.pdf>

func (*PrivateKey) [Precompute](#)

```
func (priv *PrivateKey) Precompute()
```

Precompute performs some calculations that speed up private key operations in

the future.

func (*PrivateKey) [Validate](#)

```
func (priv *PrivateKey) Validate() error
```

Validate performs basic sanity checks on the key. It returns nil if the key is valid, or else an error describing a problem.

type [PublicKey](#)

```
type PublicKey struct {  
    N *big.Int // modulus  
    E int      // public exponent  
}
```

A `PublicKey` represents the public part of an RSA key.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package sha1

```
import "crypto/sha1"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package sha1 implements the SHA1 hash algorithm as defined in RFC 3174.

Index

[Constants](#)

[func New\(\) hash.Hash](#)

Examples

[New](#)

Package files

[sha1.go](#) [sha1block.go](#)

Constants

```
const BlockSize = 64
```

The blocksize of SHA1 in bytes.

```
const Size = 20
```

The size of a SHA1 checksum in bytes.

func [New](#)

func New() hash.Hash

New returns a new hash.Hash computing the SHA1 checksum.

? Example

? Example

Code:

```
h := sha1.New()
io.WriteString(h, "His money is twice tainted: 'taint yours and 'taint mine")
fmt.Printf("% x", h.Sum(nil))
```

Output:

```
59 7f 6a 54 00 10 f9 4c 15 d7 18 06 a9 9a 2c 87 10 e7 47 bd
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package sha256

```
import "crypto/sha256"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package sha256 implements the SHA224 and SHA256 hash algorithms as defined in FIPS 180-2.

Index

Constants

[func New\(\) hash.Hash](#)

[func New224\(\) hash.Hash](#)

Package files

[sha256.go](#) [sha256block.go](#)

Constants

```
const BlockSize = 64
```

The blocksize of SHA256 and SHA224 in bytes.

```
const Size = 32
```

The size of a SHA256 checksum in bytes.

```
const Size224 = 28
```

The size of a SHA224 checksum in bytes.

func [New](#)

```
func New() hash.Hash
```

New returns a new hash.Hash computing the SHA256 checksum.

func [New224](#)

func New224() hash.Hash

New224 returns a new hash.Hash computing the SHA224 checksum.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package sha512

```
import "crypto/sha512"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package sha512 implements the SHA384 and SHA512 hash algorithms as defined in FIPS 180-2.

Index

Constants

[func New\(\) hash.Hash](#)

[func New384\(\) hash.Hash](#)

Package files

[sha512.go](#) [sha512block.go](#)

Constants

```
const BlockSize = 128
```

The blocksize of SHA512 and SHA384 in bytes.

```
const Size = 64
```

The size of a SHA512 checksum in bytes.

```
const Size384 = 48
```

The size of a SHA384 checksum in bytes.

func [New](#)

```
func New() hash.Hash
```

New returns a new hash.Hash computing the SHA512 checksum.

func [New384](#)

func New384() hash.Hash

New384 returns a new hash.Hash computing the SHA384 checksum.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package subtle

```
import "crypto/subtle"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package subtle implements functions that are often useful in cryptographic code but require careful thought to use correctly.

Index

[func ConstantTimeByteEq\(x, y uint8\) int](#)
[func ConstantTimeCompare\(x, y \[\]byte\) int](#)
[func ConstantTimeCopy\(v int, x, y \[\]byte\)](#)
[func ConstantTimeEq\(x, y int32\) int](#)
[func ConstantTimeSelect\(v, x, y int\) int](#)

Package files

[constant_time.go](#)

func ConstantTimeByteEq

```
func ConstantTimeByteEq(x, y uint8) int
```

ConstantTimeByteEq returns 1 if $x == y$ and 0 otherwise.

func ConstantTimeCompare

```
func ConstantTimeCompare(x, y []byte) int
```

ConstantTimeCompare returns 1 iff the two equal length slices, x and y, have equal contents. The time taken is a function of the length of the slices and is independent of the contents.

func ConstantTimeCopy

```
func ConstantTimeCopy(v int, x, y []byte)
```

ConstantTimeCopy copies the contents of y into x iff v == 1. If v == 0, x is left unchanged. Its behavior is undefined if v takes any other value.

func ConstantTimeEq

```
func ConstantTimeEq(x, y int32) int
```

ConstantTimeEq returns 1 if $x == y$ and 0 otherwise.

func [ConstantTimeSelect](#)

```
func ConstantTimeSelect(v, x, y int) int
```

ConstantTimeSelect returns x if v is 1 and y if v is 0. Its behavior is undefined if v takes any other value.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package tls

```
import "crypto/tls"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `tls` partially implements TLS 1.0, as specified in RFC 2246.

Index

Constants

[func Listen\(network, laddr string, config *Config\) \(net.Listener, error\)](#)

[func NewListener\(inner net.Listener, config *Config\) net.Listener](#)

[type Certificate](#)

[func LoadX509KeyPair\(certFile, keyFile string\) \(cert Certificate, err error\)](#)

[func X509KeyPair\(certPEMBlock, keyPEMBlock \[\]byte\) \(cert Certificate, err error\)](#)

[type ClientAuthType](#)

[type Config](#)

[func \(c *Config\) BuildNameToCertificate\(\)](#)

[type Conn](#)

[func Client\(conn net.Conn, config *Config\) *Conn](#)

[func Dial\(network, addr string, config *Config\) \(*Conn, error\)](#)

[func Server\(conn net.Conn, config *Config\) *Conn](#)

[func \(c *Conn\) Close\(\) error](#)

[func \(c *Conn\) ConnectionState\(\) ConnectionState](#)

[func \(c *Conn\) Handshake\(\) error](#)

[func \(c *Conn\) LocalAddr\(\) net.Addr](#)

[func \(c *Conn\) OCSPResponse\(\) \[\]byte](#)

[func \(c *Conn\) Read\(b \[\]byte\) \(n int, err error\)](#)

[func \(c *Conn\) RemoteAddr\(\) net.Addr](#)

[func \(c *Conn\) SetDeadline\(t time.Time\) error](#)

[func \(c *Conn\) SetReadDeadline\(t time.Time\) error](#)

[func \(c *Conn\) SetWriteDeadline\(t time.Time\) error](#)

[func \(c *Conn\) VerifyHostname\(host string\) error](#)

[func \(c *Conn\) Write\(b \[\]byte\) \(int, error\)](#)

[type ConnectionState](#)

Package files

[alert.go](#) [cipher](#) [suites.go](#) [common.go](#) [conn.go](#) [handshake](#) [client.go](#) [handshake](#) [messages.go](#)
[handshake](#) [server.go](#) [key](#) [agreement.go](#) [prf.go](#) [tls.go](#)

Constants

```
const (  
    TLS_RSA_WITH_RC4_128_SHA          uint16 = 0x0005  
    TLS_RSA_WITH_3DES_EDE_CBC_SHA     uint16 = 0x000a  
    TLS_RSA_WITH_AES_128_CBC_SHA      uint16 = 0x002f  
    TLS_ECDHE_RSA_WITH_RC4_128_SHA    uint16 = 0xc011  
    TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA uint16 = 0xc012  
    TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA uint16 = 0xc013  
)
```

A list of the possible cipher suite ids. Taken from

<http://www.iana.org/assignments/tls-parameters/tls-parameters.xml>

func [Listen](#)

```
func Listen(network, laddr string, config *Config) (net.Listener, error)
```

Listen creates a TLS listener accepting connections on the given network address using `net.Listen`. The configuration `config` must be non-nil and must have at least one certificate.

func [NewListener](#)

```
func NewListener(inner net.Listener, config *Config) net.Listener
```

NewListener creates a Listener which accepts connections from an inner Listener and wraps each connection with Server. The configuration config must be non-nil and must have at least one certificate.

type [Certificate](#)

```
type Certificate struct {
    Certificate [][]byte
    PrivateKey crypto.PrivateKey // supported types: *rsa.PrivateKe
    // OCSPStaple contains an optional OCSP response which will be s
    // to clients that request it.
    OCSPStaple []byte
    // Leaf is the parsed form of the leaf certificate, which may be
    // initialized using x509.ParseCertificate to reduce per-handsha
    // processing for TLS clients doing client authentication. If ni
    // leaf certificate will be parsed as needed.
    Leaf *x509.Certificate
}
```

A Certificate is a chain of one or more certificates, leaf first.

func [LoadX509KeyPair](#)

```
func LoadX509KeyPair(certFile, keyFile string) (cert Certificate, er
```

LoadX509KeyPair reads and parses a public/private key pair from a pair of files. The files must contain PEM encoded data.

func [X509KeyPair](#)

```
func X509KeyPair(certPEMBlock, keyPEMBlock []byte) (cert Certificate
```

X509KeyPair parses a public/private key pair from a pair of PEM encoded data.

type [ClientAuthType](#)

```
type ClientAuthType int
```

ClientAuthType declares the policy the server will follow for TLS Client Authentication.

```
const (  
    NoClientCert ClientAuthType = iota  
    RequestClientCert  
    RequireAnyClientCert  
    VerifyClientCertIfGiven  
    RequireAndVerifyClientCert  
)
```

type [Config](#)

```
type Config struct {
    // Rand provides the source of entropy for nonces and RSA blinding
    // If Rand is nil, TLS uses the cryptographic random reader in package
    // crypto/rand.
    Rand io.Reader

    // Time returns the current time as the number of seconds since epoch
    // If Time is nil, TLS uses time.Now.
    Time func() time.Time

    // Certificates contains one or more certificate chains
    // to present to the other side of the connection.
    // Server configurations must include at least one certificate.
    Certificates []Certificate

    // NameToCertificate maps from a certificate name to an element
    // of Certificates. Note that a certificate name can be of the form
    // '*.example.com' and so doesn't have to be a domain name as such.
    // See Config.BuildNameToCertificate
    // The nil value causes the first element of Certificates to be
    // used for all connections.
    NameToCertificate map[string]*Certificate

    // RootCAs defines the set of root certificate authorities
    // that clients use when verifying server certificates.
    // If RootCAs is nil, TLS uses the host's root CA set.
    RootCAs *x509.CertPool

    // NextProtos is a list of supported, application level protocols
    NextProtos []string

    // ServerName is included in the client's handshake to support virtual
    // hosting.
    ServerName string

    // ClientAuth determines the server's policy for
    // TLS Client Authentication. The default is NoClientCert.
    ClientAuth ClientAuthType

    // ClientCAs defines the set of root certificate authorities
    // that servers use if required to verify a client certificate
    // by the policy in ClientAuth.
    ClientCAs *x509.CertPool

    // InsecureSkipVerify controls whether a client verifies the

```

```

// server's certificate chain and host name.
// If InsecureSkipVerify is true, TLS accepts any certificate
// presented by the server and any host name in that certificate
// In this mode, TLS is susceptible to man-in-the-middle attacks
// This should be used only for testing.
InsecureSkipVerify bool

// CipherSuites is a list of supported cipher suites. If CipherS
// is nil, TLS uses a list of suites supported by the implementa
CipherSuites []uint16
}

```

A Config structure is used to configure a TLS client or server. After one has been passed to a TLS function it must not be modified.

func (*Config) [BuildNameToCertificate](#)

```
func (c *Config) BuildNameToCertificate()
```

BuildNameToCertificate parses c.Certificates and builds c.NameToCertificate from the CommonName and SubjectAlternateName fields of each of the leaf certificates.

type [Conn](#)

```
type Conn struct {  
    // contains filtered or unexported fields  
}
```

A Conn represents a secured connection. It implements the net.Conn interface.

func [Client](#)

```
func Client(conn net.Conn, config *Config) *Conn
```

Client returns a new TLS client side connection using conn as the underlying transport. Client interprets a nil configuration as equivalent to the zero configuration; see the documentation of Config for the defaults.

func [Dial](#)

```
func Dial(network, addr string, config *Config) (*Conn, error)
```

Dial connects to the given network address using net.Dial and then initiates a TLS handshake, returning the resulting TLS connection. Dial interprets a nil configuration as equivalent to the zero configuration; see the documentation of Config for the defaults.

func [Server](#)

```
func Server(conn net.Conn, config *Config) *Conn
```

Server returns a new TLS server side connection using conn as the underlying transport. The configuration config must be non-nil and must have at least one certificate.

func (***Conn**) [Close](#)

```
func (c *Conn) Close() error
```

Close closes the connection.

func (*Conn) [ConnectionState](#)

```
func (c *Conn) ConnectionState() ConnectionState
```

ConnectionState returns basic TLS details about the connection.

func (*Conn) [Handshake](#)

```
func (c *Conn) Handshake() error
```

Handshake runs the client or server handshake protocol if it has not yet been run. Most uses of this package need not call Handshake explicitly: the first Read or Write will call it automatically.

func (*Conn) [LocalAddr](#)

```
func (c *Conn) LocalAddr() net.Addr
```

LocalAddr returns the local network address.

func (*Conn) [OCSPResponse](#)

```
func (c *Conn) OCSPResponse() []byte
```

OCSPResponse returns the stapled OCSP response from the TLS server, if any. (Only valid for client connections.)

func (*Conn) [Read](#)

```
func (c *Conn) Read(b []byte) (n int, err error)
```

Read can be made to time out and return a net.Error with Timeout() == true after a fixed time limit; see SetDeadline and SetReadDeadline.

func (*Conn) [RemoteAddr](#)

```
func (c *Conn) RemoteAddr() net.Addr
```

RemoteAddr returns the remote network address.

func (*Conn) [SetDeadline](#)

```
func (c *Conn) SetDeadline(t time.Time) error
```

SetDeadline sets the read and write deadlines associated with the connection. A zero value for t means Read and Write will not time out. After a Write has timed out, the TLS state is corrupt and all future writes will return the same error.

func (*Conn) [SetReadDeadline](#)

```
func (c *Conn) SetReadDeadline(t time.Time) error
```

SetReadDeadline sets the read deadline on the underlying connection. A zero value for t means Read will not time out.

func (*Conn) [SetWriteDeadline](#)

```
func (c *Conn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline sets the write deadline on the underlying connection. A zero value for t means Write will not time out. After a Write has timed out, the TLS state is corrupt and all future writes will return the same error.

func (*Conn) [VerifyHostname](#)

```
func (c *Conn) VerifyHostname(host string) error
```

VerifyHostname checks that the peer certificate chain is valid for connecting to host. If so, it returns nil; if not, it returns an error describing the problem.

func (*Conn) [Write](#)

```
func (c *Conn) Write(b []byte) (int, error)
```

Write writes data to the connection.

type [ConnectionState](#)

```
type ConnectionState struct {
    HandshakeComplete      bool
    CipherSuite            uint16
    NegotiatedProtocol     string
    NegotiatedProtocolIsMutual bool

    // ServerName contains the server name indicated by the client,
    // (Only valid for server connections.)
    ServerName string

    // the certificate chain that was presented by the other side
    PeerCertificates []*x509.Certificate
    // the verified certificate chains built from PeerCertificates.
    VerifiedChains [][]*x509.Certificate
}
```

ConnectionState records basic TLS details about the connection.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package x509

```
import "crypto/x509"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package x509 parses X.509-encoded keys and certificates.

Index

Variables

func CreateCertificate(rand io.Reader, template, parent *Certificate, pub interface{}, priv interface{}) (cert []byte, err error)

func MarshalPKCS1PrivateKey(key *rsa.PrivateKey) []byte

func MarshalPKIXPublicKey(pub interface{}) ([]byte, error)

func ParseCRL(crlBytes []byte) (certList *pkix.CertificateList, err error)

func ParseCertificates(asn1Data []byte) ([]*Certificate, error)

func ParseDERCRL(derBytes []byte) (certList *pkix.CertificateList, err error)

func ParsePKCS1PrivateKey(der []byte) (key *rsa.PrivateKey, err error)

func ParsePKCS8PrivateKey(der []byte) (key interface{}, err error)

func ParsePKIXPublicKey(derBytes []byte) (pub interface{}, err error)

type CertPool

func NewCertPool() *CertPool

func (s *CertPool) AddCert(cert *Certificate)

func (s *CertPool) AppendCertsFromPEM(pemCerts []byte) (ok bool)

func (s *CertPool) Subjects() (res [][]byte)

type Certificate

func ParseCertificate(asn1Data []byte) (*Certificate, error)

func (c *Certificate) CheckCRLSignature(crl *pkix.CertificateList) (err error)

func (c *Certificate) CheckSignature(algo SignatureAlgorithm, signed, signature []byte) (err error)

func (c *Certificate) CheckSignatureFrom(parent *Certificate) (err error)

func (c *Certificate) CreateCRL(rand io.Reader, priv interface{}, revokedCerts []pkix.RevokedCertificate, now, expiry time.Time) (crlBytes []byte, err error)

func (c *Certificate) Equal(other *Certificate) bool

func (c *Certificate) Verify(opts VerifyOptions) (chains [][]*Certificate, err error)

func (c *Certificate) VerifyHostname(h string) error

type CertificateInvalidError

func (e CertificateInvalidError) Error() string

type ConstraintViolationError

func (ConstraintViolationError) Error() string

[type ExtKeyUsage](#)
[type HostnameError](#)
 [func \(h HostnameError\) Error\(\) string](#)
[type InvalidReason](#)
[type KeyUsage](#)
[type PublicKeyAlgorithm](#)
[type SignatureAlgorithm](#)
[type UnhandledCriticalExtension](#)
 [func \(h UnhandledCriticalExtension\) Error\(\) string](#)
[type UnknownAuthorityError](#)
 [func \(e UnknownAuthorityError\) Error\(\) string](#)
[type VerifyOptions](#)

Package files

[cert_pool.go](#) [pkcs1.go](#) [pkcs8.go](#) [root.go](#) [root_unix.go](#) [verify.go](#) [x509.go](#)

Variables

```
var ErrUnsupportedAlgorithm = errors.New("crypto/x509: cannot verify
```

ErrUnsupportedAlgorithm results from attempting to perform an operation that involves algorithms that are not currently implemented.

func [CreateCertificate](#)

```
func CreateCertificate(rand io.Reader, template, parent *Certificate
```

CreateCertificate creates a new certificate based on a template. The following members of template are used: SerialNumber, Subject, NotBefore, NotAfter, KeyUsage, BasicConstraintsValid, IsCA, MaxPathLen, SubjectKeyId, DNSNames, PermittedDNSDomainsCritical, PermittedDNSDomains.

The certificate is signed by parent. If parent is equal to template then the certificate is self-signed. The parameter pub is the public key of the signee and priv is the private key of the signer.

The returned slice is the certificate in DER encoding.

The only supported key type is RSA (*rsa.PublicKey for pub, *rsa.PrivateKey for priv).

func [MarshalPKCS1PrivateKey](#)

```
func MarshalPKCS1PrivateKey(key *rsa.PrivateKey) []byte
```

MarshalPKCS1PrivateKey converts a private key to ASN.1 DER encoded form.

func [MarshalPKIXPublicKey](#)

```
func MarshalPKIXPublicKey(pub interface{}) ([]byte, error)
```

MarshalPKIXPublicKey serialises a public key to DER-encoded PKIX format.

func [ParseCRL](#)

```
func ParseCRL(crlBytes []byte) (certList *pkix.CertificateList, err
```

ParseCRL parses a CRL from the given bytes. It's often the case that PEM encoded CRLs will appear where they should be DER encoded, so this function will transparently handle PEM encoding as long as there isn't any leading garbage.

func [ParseCertificates](#)

```
func ParseCertificates(asn1Data []byte) ([]*Certificate, error)
```

ParseCertificates parses one or more certificates from the given ASN.1 DER data. The certificates must be concatenated with no intermediate padding.

func ParseDERCRL

```
func ParseDERCRL(derBytes []byte) (certList *pkix.CertificateList, e
```

ParseDERCRL parses a DER encoded CRL from the given bytes.

func [ParsePKCS1PrivateKey](#)

```
func ParsePKCS1PrivateKey(der []byte) (key *rsa.PrivateKey, err error)
```

ParsePKCS1PrivateKey returns an RSA private key from its ASN.1 PKCS#1 DER encoded form.

func [ParsePKCS8PrivateKey](#)

func ParsePKCS8PrivateKey(der []byte) (key interface{}, err error)

ParsePKCS8PrivateKey parses an unencrypted, PKCS#8 private key. See <http://www.rsa.com/rsalabs/node.asp?id=2130>

func [ParsePKIXPublicKey](#)

```
func ParsePKIXPublicKey(derBytes []byte) (pub interface{}, err error
```

ParsePKIXPublicKey parses a DER encoded public key. These values are typically found in PEM blocks with "BEGIN PUBLIC KEY".

type [CertPool](#)

```
type CertPool struct {  
    // contains filtered or unexported fields  
}
```

CertPool is a set of certificates.

func [NewCertPool](#)

```
func NewCertPool() *CertPool
```

NewCertPool returns a new, empty CertPool.

func (***CertPool**) [AddCert](#)

```
func (s *CertPool) AddCert(cert *Certificate)
```

AddCert adds a certificate to a pool.

func (***CertPool**) [AppendCertsFromPEM](#)

```
func (s *CertPool) AppendCertsFromPEM(pemCerts []byte) (ok bool)
```

AppendCertsFromPEM attempts to parse a series of PEM encoded certificates. It appends any certificates found to s and returns true if any certificates were successfully parsed.

On many Linux systems, /etc/ssl/cert.pem will contain the system wide set of root CAs in a format suitable for this function.

func (***CertPool**) [Subjects](#)

```
func (s *CertPool) Subjects() (res [][]byte)
```

Subjects returns a list of the DER-encoded subjects of all of the certificates in the pool.

type [Certificate](#)

```
type Certificate struct {
    Raw []byte // Complete ASN.1 DER content (certificate)
    RawTBSCertificate []byte // Certificate part of raw ASN.1
    RawSubjectPublicKeyInfo []byte // DER encoded SubjectPublicKeyInfo
    RawSubject []byte // DER encoded Subject
    RawIssuer []byte // DER encoded Issuer

    Signature []byte
    SignatureAlgorithm SignatureAlgorithm

    PublicKeyAlgorithm PublicKeyAlgorithm
    PublicKey interface{}

    Version int
    SerialNumber *big.Int
    Issuer pkix.Name
    Subject pkix.Name
    NotBefore, NotAfter time.Time // Validity bounds.
    KeyUsage KeyUsage

    ExtKeyUsage []ExtKeyUsage // Sequence of extended key usages
    UnknownExtKeyUsage []asn1.ObjectIdentifier // Encountered extended key usages

    BasicConstraintsValid bool // if true then the next two fields are present
    IsCA bool
    MaxPathLen int

    SubjectKeyId []byte
    AuthorityKeyId []byte

    // Subject Alternate Name values
    DNSNames []string
    EmailAddresses []string

    // Name constraints
    PermittedDNSDomainsCritical bool // if true then the name constraints are critical
    PermittedDNSDomains []string

    PolicyIdentifiers []asn1.ObjectIdentifier
}
```

A Certificate represents an X.509 certificate.

func [ParseCertificate](#)

```
func ParseCertificate(asn1Data []byte) (*Certificate, error)
```

ParseCertificate parses a single certificate from the given ASN.1 DER data.

func (*Certificate) [CheckCRLSignature](#)

```
func (c *Certificate) CheckCRLSignature(crl *pkix.CertificateList) (
```

CheckCRLSignature checks that the signature in crl is from c.

func (*Certificate) [CheckSignature](#)

```
func (c *Certificate) CheckSignature(algo SignatureAlgorithm, signed
```

CheckSignature verifies that signature is a valid signature over signed from c's public key.

func (*Certificate) [CheckSignatureFrom](#)

```
func (c *Certificate) CheckSignatureFrom(parent *Certificate) (err e
```

CheckSignatureFrom verifies that the signature on c is a valid signature from parent.

func (*Certificate) [CreateCRL](#)

```
func (c *Certificate) CreateCRL(rand io.Reader, priv interface{}, re
```

CreateCRL returns a DER encoded CRL, signed by this Certificate, that contains the given list of revoked certificates.

The only supported key type is RSA (*rsa.PrivateKey for priv).

func (*Certificate) [Equal](#)

```
func (c *Certificate) Equal(other *Certificate) bool
```

func (*Certificate) [Verify](#)

```
func (c *Certificate) Verify(opts VerifyOptions) (chains [][]*Certif
```

Verify attempts to verify `c` by building one or more chains from `c` to a certificate in `opts.Roots`, using certificates in `opts.Intermediates` if needed. If successful, it returns one or more chains where the first element of the chain is `c` and the last element is from `opts.Roots`.

WARNING: this doesn't do any revocation checking.

func (*Certificate) [VerifyHostname](#)

```
func (c *Certificate) VerifyHostname(h string) error
```

`VerifyHostname` returns `nil` if `c` is a valid certificate for the named host. Otherwise it returns an error describing the mismatch.

type [CertificateInvalidError](#)

```
type CertificateInvalidError struct {  
    Cert    *Certificate  
    Reason  InvalidReason  
}
```

CertificateInvalidError results when an odd error occurs. Users of this library probably want to handle all these errors uniformly.

func (CertificateInvalidError) [Error](#)

```
func (e CertificateInvalidError) Error() string
```

type [ConstraintViolationError](#)

```
type ConstraintViolationError struct{}
```

ConstraintViolationError results when a requested usage is not permitted by a certificate. For example: checking a signature when the public key isn't a certificate signing key.

func (ConstraintViolationError) [Error](#)

```
func (ConstraintViolationError) Error() string
```

type [ExtKeyUsage](#)

```
type ExtKeyUsage int
```

ExtKeyUsage represents an extended set of actions that are valid for a given key. Each of the ExtKeyUsage* constants define a unique action.

```
const (  
    ExtKeyUsageAny ExtKeyUsage = iota  
    ExtKeyUsageServerAuth  
    ExtKeyUsageClientAuth  
    ExtKeyUsageCodeSigning  
    ExtKeyUsageEmailProtection  
    ExtKeyUsageTimeStamping  
    ExtKeyUsageOCSPSigning  
)
```

type [HostnameError](#)

```
type HostnameError struct {  
    Certificate *Certificate  
    Host       string  
}
```

HostnameError results when the set of authorized names doesn't match the requested name.

func ([HostnameError](#)) [Error](#)

```
func (h HostnameError) Error() string
```

type InvalidReason

```
type InvalidReason int
```

```
const (  
    // NotAuthorizedToSign results when a certificate is signed by a  
    // which isn't marked as a CA certificate.  
    NotAuthorizedToSign InvalidReason = iota  
    // Expired results when a certificate has expired, based on the  
    // given in the VerifyOptions.  
    Expired  
    // CANotAuthorizedForThisName results when an intermediate or ro  
    // certificate has a name constraint which doesn't include the n  
    // being checked.  
    CANotAuthorizedForThisName  
    // TooManyIntermediates results when a path length constraint is  
    // violated.  
    TooManyIntermediates  
)
```

type [KeyUsage](#)

```
type KeyUsage int
```

KeyUsage represents the set of actions that are valid for a given key. It's a bitmap of the KeyUsage* constants.

```
const (  
    KeyUsageDigitalSignature KeyUsage = 1 << iota  
    KeyUsageContentCommitment  
    KeyUsageKeyEncipherment  
    KeyUsageDataEncipherment  
    KeyUsageKeyAgreement  
    KeyUsageCertSign  
    KeyUsageCRLSign  
    KeyUsageEncipherOnly  
    KeyUsageDecipherOnly  
)
```

type PublicKeyAlgorithm

```
type PublicKeyAlgorithm int
```

```
const (  
    UnknownPublicKeyAlgorithm PublicKeyAlgorithm = iota  
    RSA  
    DSA  
)
```

type [SignatureAlgorithm](#)

```
type SignatureAlgorithm int
```

```
const (  
    UnknownSignatureAlgorithm SignatureAlgorithm = iota  
    MD2WithRSA  
    MD5WithRSA  
    SHA1WithRSA  
    SHA256WithRSA  
    SHA384WithRSA  
    SHA512WithRSA  
    DSAWithSHA1  
    DSAWithSHA256  
)
```

type [UnhandledCriticalExtension](#)

```
type UnhandledCriticalExtension struct{}
```

func (UnhandledCriticalExtension) [Error](#)

```
func (h UnhandledCriticalExtension) Error() string
```

type [UnknownAuthorityError](#)

```
type UnknownAuthorityError struct {  
    // contains filtered or unexported fields  
}
```

UnknownAuthorityError results when the certificate issuer is unknown

func (UnknownAuthorityError) [Error](#)

```
func (e UnknownAuthorityError) Error() string
```

type [VerifyOptions](#)

```
type VerifyOptions struct {
    DNSName      string
    Intermediates *CertPool
    Roots        *CertPool // if nil, the system roots are used
    CurrentTime  time.Time // if zero, the current time is used
}
```

VerifyOptions contains parameters for Certificate.Verify. It's a structure because other PKIX verification APIs have ended up needing many options.

Subdirectories

Name	Synopsis
------	----------

pkix	Package pkix contains shared, low level structures used for ASN.1 parsing and serialization of X.509 certificates, CRL and OCSP.
----------------------	--

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package pkix

```
import "crypto/x509/pkix"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package pkix contains shared, low level structures used for ASN.1 parsing and serialization of X.509 certificates, CRL and OCSP.

Index

[type AlgorithmIdentifier](#)

[type AttributeTypeAndValue](#)

[type CertificateList](#)

[func \(certList *CertificateList\) HasExpired\(now time.Time\) bool](#)

[type Extension](#)

[type Name](#)

[func \(n *Name\) FillFromRDNSequence\(rdns *RDNSequence\)](#)

[func \(n Name\) ToRDNSequence\(\) \(ret RDNSequence\)](#)

[type RDNSequence](#)

[type RelativeDistinguishedNameSET](#)

[type RevokedCertificate](#)

[type TBSCertificateList](#)

Package files

[pkix.go](#)

type [AlgorithmIdentifier](#)

```
type AlgorithmIdentifier struct {  
    Algorithm asn1.ObjectIdentifier  
    Parameters asn1.RawValue `asn1:"optional"`  
}
```

AlgorithmIdentifier represents the ASN.1 structure of the same name. See RFC 5280, section 4.1.1.2.

type AttributeTypeAndValue

```
type AttributeTypeAndValue struct {  
    Type  asn1.ObjectIdentifier  
    Value interface{}}
```

AttributeTypeAndValue mirrors the ASN.1 structure of the same name in <http://tools.ietf.org/html/rfc5280#section-4.1.2.4>

type [CertificateList](#)

```
type CertificateList struct {  
    TBSCertList      TBSCertificateList  
    SignatureAlgorithm AlgorithmIdentifier  
    SignatureValue   asn1.BitString  
}
```

CertificateList represents the ASN.1 structure of the same name. See RFC 5280, section 5.1. Use Certificate.CheckCRLSignature to verify the signature.

func (*CertificateList) [HasExpired](#)

```
func (certList *CertificateList) HasExpired(now time.Time) bool
```

HasExpired returns true iff now is past the expiry time of certList.

type [Extension](#)

```
type Extension struct {
    Id          asn1.ObjectIdentifier
    Critical    bool `asn1:"optional"`
    Value      []byte
}
```

Extension represents the ASN.1 structure of the same name. See RFC 5280, section 4.2.

type [Name](#)

```
type Name struct {
    Country, Organization, OrganizationalUnit []string
    Locality, Province                       []string
    StreetAddress, PostalCode                []string
    SerialNumber, CommonName                 string
    Names []AttributeTypeAndValue
}
```

Name represents an X.509 distinguished name. This only includes the common elements of a DN. Additional elements in the name are ignored.

func (*Name) [FillFromRDNSequence](#)

```
func (n *Name) FillFromRDNSequence(rdns *RDNSequence)
```

func (Name) [ToRDNSequence](#)

```
func (n Name) ToRDNSequence() (ret RDNSequence)
```

type [RDNSequence](#)

type RDNSequence []RelativeDistinguishedNameSET

type RelativeDistinguishedNameSET

type RelativeDistinguishedNameSET []AttributeTypeAndValue

type [RevokedCertificate](#)

```
type RevokedCertificate struct {  
    SerialNumber    *big.Int  
    RevocationTime  time.Time  
    Extensions      []Extension `asn1:"optional"`  
}
```

RevokedCertificate represents the ASN.1 structure of the same name. See RFC 5280, section 5.1.

type [TBSCertificateList](#)

```
type TBSCertificateList struct {
    Raw          asn1.RawContent
    Version      int `asn1:"optional,default:2"`
    Signature    AlgorithmIdentifier
    Issuer       RDNSequence
    ThisUpdate   time.Time
    NextUpdate   time.Time
    RevokedCertificates []RevokedCertificate `asn1:"optional"`
    Extensions   []Extension           `asn1:"tag:0,optional,e`
}
```

TBSCertificateList represents the ASN.1 structure of the same name. See RFC 5280, section 5.1.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg/database

Name	Synopsis
sql	Package sql provides a generic interface around SQL (or SQL-like) databases.
driver	Package driver defines interfaces to be implemented by database drivers as used by package sql.

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package sql

```
import "database/sql"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package sql provides a generic interface around SQL (or SQL-like) databases.

Index

Variables

[func Register\(name string, driver driver.Driver\)](#)

type DB

[func Open\(driverName, dataSourceName string\) \(*DB, error\)](#)

[func \(db *DB\) Begin\(\) \(*Tx, error\)](#)

[func \(db *DB\) Close\(\) error](#)

[func \(db *DB\) Driver\(\) driver.Driver](#)

[func \(db *DB\) Exec\(query string, args ...interface{}\) \(Result, error\)](#)

[func \(db *DB\) Prepare\(query string\) \(*Stmt, error\)](#)

[func \(db *DB\) Query\(query string, args ...interface{}\) \(*Rows, error\)](#)

[func \(db *DB\) QueryRow\(query string, args ...interface{}\) *Row](#)

type NullBool

[func \(n *NullBool\) Scan\(value interface{}\) error](#)

[func \(n NullBool\) Value\(\) \(driver.Value, error\)](#)

type NullFloat64

[func \(n *NullFloat64\) Scan\(value interface{}\) error](#)

[func \(n NullFloat64\) Value\(\) \(driver.Value, error\)](#)

type NullInt64

[func \(n *NullInt64\) Scan\(value interface{}\) error](#)

[func \(n NullInt64\) Value\(\) \(driver.Value, error\)](#)

type NullString

[func \(ns *NullString\) Scan\(value interface{}\) error](#)

[func \(ns NullString\) Value\(\) \(driver.Value, error\)](#)

type RawBytes

type Result

type Row

[func \(r *Row\) Scan\(dest ...interface{}\) error](#)

type Rows

[func \(rs *Rows\) Close\(\) error](#)

[func \(rs *Rows\) Columns\(\) \(\[\]string, error\)](#)

[func \(rs *Rows\) Err\(\) error](#)

[func \(rs *Rows\) Next\(\) bool](#)

[func \(rs *Rows\) Scan\(dest ...interface{}\) error](#)

type Scanner

type Stmt

[func \(s *Stmt\) Close\(\) error](#)
[func \(s *Stmt\) Exec\(args ...interface{}\) \(Result, error\)](#)
[func \(s *Stmt\) Query\(args ...interface{}\) \(*Rows, error\)](#)
[func \(s *Stmt\) QueryRow\(args ...interface{}\) *Row](#)

[type Tx](#)

[func \(tx *Tx\) Commit\(\) error](#)
[func \(tx *Tx\) Exec\(query string, args ...interface{}\) \(Result, error\)](#)
[func \(tx *Tx\) Prepare\(query string\) \(*Stmt, error\)](#)
[func \(tx *Tx\) Query\(query string, args ...interface{}\) \(*Rows, error\)](#)
[func \(tx *Tx\) QueryRow\(query string, args ...interface{}\) *Row](#)
[func \(tx *Tx\) Rollback\(\) error](#)
[func \(tx *Tx\) Stmt\(stmt *Stmt\) *Stmt](#)

Package files

[convert.go](#) [sql.go](#)

Variables

```
var ErrNoRows = errors.New("sql: no rows in result set")
```

ErrNoRows is returned by Scan when QueryRow doesn't return a row. In such a case, QueryRow returns a placeholder *Row value that defers this error until a Scan.

```
var ErrTxDone = errors.New("sql: Transaction has already been commit
```

func [Register](#)

```
func Register(name string, driver driver.Driver)
```

Register makes a database driver available by the provided name. If Register is called twice with the same name or if driver is nil, it panics.

type [DB](#)

```
type DB struct {  
    // contains filtered or unexported fields  
}
```

DB is a database handle. It's safe for concurrent use by multiple goroutines.

If the underlying database driver has the concept of a connection and per-connection session state, the sql package manages creating and freeing connections automatically, including maintaining a free pool of idle connections. If observing session state is required, either do not share a *DB between multiple concurrent goroutines or create and observe all state only within a transaction. Once DB.Open is called, the returned Tx is bound to a single isolated connection. Once Tx.Commit or Tx.Rollback is called, that connection is returned to DB's idle connection pool.

func [Open](#)

```
func Open(driverName, dataSourceName string) (*DB, error)
```

Open opens a database specified by its database driver name and a driver-specific data source name, usually consisting of at least a database name and connection information.

Most users will open a database via a driver-specific connection helper function that returns a *DB.

func (*DB) [Begin](#)

```
func (db *DB) Begin() (*Tx, error)
```

Begin starts a transaction. The isolation level is dependent on the driver.

func (*DB) [Close](#)

```
func (db *DB) Close() error
```

Close closes the database, releasing any open resources.

func (*DB) [Driver](#)

```
func (db *DB) Driver() driver.Driver
```

Driver returns the database's underlying driver.

func (*DB) [Exec](#)

```
func (db *DB) Exec(query string, args ...interface{}) (Result, error)
```

Exec executes a query without returning any rows.

func (*DB) [Prepare](#)

```
func (db *DB) Prepare(query string) (*Stmt, error)
```

Prepare creates a prepared statement for later execution.

func (*DB) [Query](#)

```
func (db *DB) Query(query string, args ...interface{}) (*Rows, error)
```

Query executes a query that returns rows, typically a SELECT.

func (*DB) [QueryRow](#)

```
func (db *DB) QueryRow(query string, args ...interface{}) *Row
```

QueryRow executes a query that is expected to return at most one row. QueryRow always return a non-nil value. Errors are deferred until Row's Scan method is called.

type [NullBool](#)

```
type NullBool struct {  
    Bool bool  
    Valid bool // Valid is true if Bool is not NULL  
}
```

NullBool represents a bool that may be null. NullBool implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (***NullBool**) [Scan](#)

```
func (n *NullBool) Scan(value interface{}) error
```

Scan implements the Scanner interface.

func (**NullBool**) [Value](#)

```
func (n NullBool) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type [NullFloat64](#)

```
type NullFloat64 struct {  
    Float64 float64  
    Valid    bool // Valid is true if Float64 is not NULL  
}
```

NullFloat64 represents a float64 that may be null. NullFloat64 implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (*NullFloat64) [Scan](#)

```
func (n *NullFloat64) Scan(value interface{}) error
```

Scan implements the Scanner interface.

func (NullFloat64) [Value](#)

```
func (n NullFloat64) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type [NullInt64](#)

```
type NullInt64 struct {  
    Int64 int64  
    Valid bool // Valid is true if Int64 is not NULL  
}
```

NullInt64 represents an int64 that may be null. NullInt64 implements the Scanner interface so it can be used as a scan destination, similar to NullString.

func (***NullInt64**) [Scan](#)

```
func (n *NullInt64) Scan(value interface{}) error
```

Scan implements the Scanner interface.

func (**NullInt64**) [Value](#)

```
func (n NullInt64) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type [NullString](#)

```
type NullString struct {
    String string
    Valid bool // Valid is true if String is not NULL
}
```

NullString represents a string that may be null. NullString implements the Scanner interface so it can be used as a scan destination:

```
var s NullString
err := db.QueryRow("SELECT name FROM foo WHERE id=?", id).Scan(&s)
...
if s.Valid {
    // use s.String
} else {
    // NULL value
}
```

func (***NullString**) [Scan](#)

```
func (ns *NullString) Scan(value interface{}) error
```

Scan implements the Scanner interface.

func (**NullString**) [Value](#)

```
func (ns NullString) Value() (driver.Value, error)
```

Value implements the driver Valuer interface.

type [RawBytes](#)

```
type RawBytes []byte
```

RawBytes is a byte slice that holds a reference to memory owned by the database itself. After a Scan into a RawBytes, the slice is only valid until the next call to Next, Scan, or Close.

type [Result](#)

```
type Result interface {  
    LastInsertId() (int64, error)  
    RowsAffected() (int64, error)  
}
```

A Result summarizes an executed SQL command.

type [Row](#)

```
type Row struct {  
    // contains filtered or unexported fields  
}
```

Row is the result of calling QueryRow to select a single row.

func (*Row) [Scan](#)

```
func (r *Row) Scan(dest ...interface{}) error
```

Scan copies the columns from the matched row into the values pointed at by dest. If more than one row matches the query, Scan uses the first row and discards the rest. If no row matches the query, Scan returns ErrNoRows.

type [Rows](#)

```
type Rows struct {  
    // contains filtered or unexported fields  
}
```

Rows is the result of a query. Its cursor starts before the first row of the result set. Use Next to advance through the rows:

```
rows, err := db.Query("SELECT ...")  
...  
for rows.Next() {  
    var id int  
    var name string  
    err = rows.Scan(&id, &name)  
    ...  
}  
err = rows.Err() // get any error encountered during iteration  
...
```

func (***Rows**) [Close](#)

```
func (rs *Rows) Close() error
```

Close closes the Rows, preventing further enumeration. If the end is encountered, the Rows are closed automatically. Close is idempotent.

func (***Rows**) [Columns](#)

```
func (rs *Rows) Columns() ([]string, error)
```

Columns returns the column names. Columns returns an error if the rows are closed, or if the rows are from QueryRow and there was a deferred error.

func (***Rows**) [Err](#)

```
func (rs *Rows) Err() error
```

Err returns the error, if any, that was encountered during iteration.

func (***Rows**) [Next](#)

```
func (rs *Rows) Next() bool
```

Next prepares the next result row for reading with the Scan method. It returns true on success, false if there is no next result row. Every call to Scan, even the first one, must be preceded by a call to Next.

func (*Rows) [Scan](#)

```
func (rs *Rows) Scan(dest ...interface{}) error
```

Scan copies the columns in the current row into the values pointed at by dest.

If an argument has type `[]byte`, Scan saves in that argument a copy of the corresponding data. The copy is owned by the caller and can be modified and held indefinitely. The copy can be avoided by using an argument of type `*RawBytes` instead; see the documentation for `RawBytes` for restrictions on its use.

If an argument has type `*interface{}`, Scan copies the value provided by the underlying driver without conversion. If the value is of type `[]byte`, a copy is made and the caller owns the result.

type [Scanner](#)

```
type Scanner interface {
    // Scan assigns a value from a database driver.
    //
    // The src value will be of one of the following restricted
    // set of types:
    //
    //     int64
    //     float64
    //     bool
    //     []byte
    //     string
    //     time.Time
    //     nil - for NULL values
    //
    // An error should be returned if the value can not be stored
    // without loss of information.
    Scan(src interface{}) error
}
```

Scanner is an interface used by Scan.

type [Stmt](#)

```
type Stmt struct {  
    // contains filtered or unexported fields  
}
```

Stmt is a prepared statement. Stmt is safe for concurrent use by multiple goroutines.

func (*Stmt) [Close](#)

```
func (s *Stmt) Close() error
```

Close closes the statement.

func (*Stmt) [Exec](#)

```
func (s *Stmt) Exec(args ...interface{}) (Result, error)
```

Exec executes a prepared statement with the given arguments and returns a Result summarizing the effect of the statement.

func (*Stmt) [Query](#)

```
func (s *Stmt) Query(args ...interface{}) (*Rows, error)
```

Query executes a prepared query statement with the given arguments and returns the query results as a *Rows.

func (*Stmt) [QueryRow](#)

```
func (s *Stmt) QueryRow(args ...interface{}) *Row
```

QueryRow executes a prepared query statement with the given arguments. If an error occurs during the execution of the statement, that error will be returned by a call to Scan on the returned *Row, which is always non-nil. If the query selects no rows, the *Row's Scan will return ErrNoRows. Otherwise, the *Row's Scan scans the first selected row and discards the rest.

Example usage:

```
var name string
err := nameByUserIdStmt.QueryRow(id).Scan(&name)
```

type Tx

```
type Tx struct {  
    // contains filtered or unexported fields  
}
```

Tx is an in-progress database transaction.

A transaction must end with a call to Commit or Rollback.

After a call to Commit or Rollback, all operations on the transaction fail with ErrTxDone.

func (*Tx) Commit

```
func (tx *Tx) Commit() error
```

Commit commits the transaction.

func (*Tx) Exec

```
func (tx *Tx) Exec(query string, args ...interface{}) (Result, error)
```

Exec executes a query that doesn't return rows. For example: an INSERT and UPDATE.

func (*Tx) Prepare

```
func (tx *Tx) Prepare(query string) (*Stmt, error)
```

Prepare creates a prepared statement for use within a transaction.

The returned statement operates within the transaction and can no longer be used once the transaction has been committed or rolled back.

To use an existing prepared statement on this transaction, see Tx.Stmt.

func (*Tx) Query

```
func (tx *Tx) Query(query string, args ...interface{}) (*Rows, error
```

Query executes a query that returns rows, typically a SELECT.

func (*Tx) [QueryRow](#)

```
func (tx *Tx) QueryRow(query string, args ...interface{}) *Row
```

QueryRow executes a query that is expected to return at most one row. QueryRow always return a non-nil value. Errors are deferred until Row's Scan method is called.

func (*Tx) [Rollback](#)

```
func (tx *Tx) Rollback() error
```

Rollback aborts the transaction.

func (*Tx) [Stmt](#)

```
func (tx *Tx) Stmt(stmt *Stmt) *Stmt
```

Stmt returns a transaction-specific prepared statement from an existing statement.

Example:

```
updateMoney, err := db.Prepare("UPDATE balance SET money=money+? WHE  
...  
tx, err := db.Begin()  
...  
res, err := tx.Stmt(updateMoney).Exec(123.45, 98293203)
```

Subdirectories

Name **Synopsis**

[driver](#) Package driver defines interfaces to be implemented by database drivers as used by package sql.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package driver

```
import "database/sql/driver"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package driver defines interfaces to be implemented by database drivers as used by package sql.

Most code should use package sql.

Index

Variables

[func IsScanValue\(v interface{}\) bool](#)

[func IsValue\(v interface{}\) bool](#)

[type ColumnConverter](#)

[type Conn](#)

[type Driver](#)

[type Execer](#)

[type NotNull](#)

[func \(n NotNull\) ConvertValue\(v interface{}\) \(Value, error\)](#)

[type Null](#)

[func \(n Null\) ConvertValue\(v interface{}\) \(Value, error\)](#)

[type Result](#)

[type Rows](#)

[type RowsAffected](#)

[func \(RowsAffected\) LastInsertId\(\) \(int64, error\)](#)

[func \(v RowsAffected\) RowsAffected\(\) \(int64, error\)](#)

[type Stmt](#)

[type Tx](#)

[type Value](#)

[type ValueConverter](#)

[type Valuer](#)

Package files

[driver.go](#) [types.go](#)

Variables

```
var Bool boolType
```

Bool is a ValueConverter that converts input values to bools.

The conversion rules are:

- booleans are returned unchanged
- for integer types,
 - 1 is true
 - 0 is false,
 - other integers are an error
- for strings and []byte, same rules as strconv.ParseBool
- all other types are an error

```
var DefaultParameterConverter defaultConverter
```

DefaultParameterConverter is the default implementation of ValueConverter that's used when a Stmt doesn't implement ColumnConverter.

DefaultParameterConverter returns the given value directly if IsValue(value). Otherwise integer type are converted to int64, floats to float64, and strings to []byte. Other types are an error.

```
var ErrBadConn = errors.New("driver: bad connection")
```

ErrBadConn should be returned by a driver to signal to the sql package that a driver.Conn is in a bad state (such as the server having earlier closed the connection) and the sql package should retry on a new connection.

To prevent duplicate operations, ErrBadConn should NOT be returned if there's a possibility that the database server might have performed the operation. Even if the server sends back an error, you shouldn't return ErrBadConn.

```
var ErrSkip = errors.New("driver: skip fast-path; continue as if uni
```

ErrSkip may be returned by some optional interfaces' methods to indicate at runtime that the fast path is unavailable and the sql package should continue as if the optional interface was not implemented. ErrSkip is only supported where explicitly documented.

```
var Int32 int32Type
```

Int32 is a ValueConverter that converts input values to int64, respecting the limits of an int32 value.

```
var ResultNoRows noRows
```

ResultNoRows is a pre-defined Result for drivers to return when a DDL command (such as a CREATE TABLE) succeeds. It returns an error for both LastInsertId and RowsAffected.

```
var String stringType
```

String is a ValueConverter that converts its input to a string. If the value is already a string or []byte, it's unchanged. If the value is of another type, conversion to string is done with `fmt.Sprintf("%v", v)`.

func [IsScanValue](#)

```
func IsScanValue(v interface{}) bool
```

IsScanValue reports whether v is a valid Value scan type. Unlike IsValue, IsScanValue does not permit the string type.

func [IsValue](#)

```
func IsValue(v interface{}) bool
```

IsValue reports whether v is a valid Value parameter type. Unlike IsScanValue, IsValue permits the string type.

type ColumnConverter

```
type ColumnConverter interface {  
    // ColumnConverter returns a ValueConverter for the provided  
    // column index. If the type of a specific column isn't known  
    // or shouldn't be handled specially, DefaultValueConverter  
    // can be returned.  
    ColumnConverter(idx int) ValueConverter  
}
```

ColumnConverter may be optionally implemented by Stmt if the the statement is aware of its own columns' types and can convert from any type to a driver Value.

type Conn

```
type Conn interface {
    // Prepare returns a prepared statement, bound to this connectio
    Prepare(query string) (Stmt, error)

    // Close invalidates and potentially stops any current
    // prepared statements and transactions, marking this
    // connection as no longer in use.
    //
    // Because the sql package maintains a free pool of
    // connections and only calls Close when there's a surplus of
    // idle connections, it shouldn't be necessary for drivers to
    // do their own connection caching.
    Close() error

    // Begin starts and returns a new transaction.
    Begin() (Tx, error)
}
```

Conn is a connection to a database. It is not used concurrently by multiple goroutines.

Conn is assumed to be stateful.

type [Driver](#)

```
type Driver interface {
    // Open returns a new connection to the database.
    // The name is a string in a driver-specific format.
    //
    // Open may return a cached connection (one previously
    // closed), but doing so is unnecessary; the sql package
    // maintains a pool of idle connections for efficient re-use.
    //
    // The returned connection is only used by one goroutine at a
    // time.
    Open(name string) (Conn, error)
}
```

Driver is the interface that must be implemented by a database driver.

type [Execer](#)

```
type Execer interface {  
    Exec(query string, args []Value) (Result, error)  
}
```

Execer is an optional interface that may be implemented by a Conn.

If a Conn does not implement Execer, the db package's DB.Exec will first prepare a query, execute the statement, and then close the statement.

Exec may return ErrSkip.

type [NotNull](#)

```
type NotNull struct {  
    Converter ValueConverter  
}
```

NotNull is a type that implements ValueConverter by disallowing nil values but otherwise delegating to another ValueConverter.

func (NotNull) [ConvertValue](#)

```
func (n NotNull) ConvertValue(v interface{}) (Value, error)
```

type [Null](#)

```
type Null struct {  
    Converter ValueConverter  
}
```

Null is a type that implements ValueConverter by allowing nil values but otherwise delegating to another ValueConverter.

func (Null) [ConvertValue](#)

```
func (n Null) ConvertValue(v interface{}) (Value, error)
```

type [Result](#)

```
type Result interface {  
    // LastInsertId returns the database's auto-generated ID  
    // after, for example, an INSERT into a table with primary  
    // key.  
    LastInsertId() (int64, error)  
  
    // RowsAffected returns the number of rows affected by the  
    // query.  
    RowsAffected() (int64, error)  
}
```

Result is the result of a query execution.

type Rows

```
type Rows interface {
    // Columns returns the names of the columns. The number of
    // columns of the result is inferred from the length of the
    // slice. If a particular column name isn't known, an empty
    // string should be returned for that entry.
    Columns() []string

    // Close closes the rows iterator.
    Close() error

    // Next is called to populate the next row of data into
    // the provided slice. The provided slice will be the same
    // size as the Columns() are wide.
    //
    // The dest slice may be populated only with
    // a driver Value type, but excluding string.
    // All string values must be converted to []byte.
    //
    // Next should return io.EOF when there are no more rows.
    Next(dest []Value) error
}
```

Rows is an iterator over an executed query's results.

type [RowsAffected](#)

type RowsAffected int64

RowsAffected implements Result for an INSERT or UPDATE operation which mutates a number of rows.

func (RowsAffected) [LastInsertId](#)

func (RowsAffected) LastInsertId() (int64, error)

func (RowsAffected) [RowsAffected](#)

func (v RowsAffected) RowsAffected() (int64, error)

type Stmt

```
type Stmt interface {
    // Close closes the statement.
    //
    // Closing a statement should not interrupt any outstanding
    // query created from that statement. That is, the following
    // order of operations is valid:
    //
    // * create a driver statement
    // * call Query on statement, returning Rows
    // * close the statement
    // * read from Rows
    //
    // If closing a statement invalidates currently-running
    // queries, the final step above will incorrectly fail.
    //
    // TODO(bradfitz): possibly remove the restriction above, if
    // enough driver authors object and find it complicates their
    // code too much. The sql package could be smarter about
    // recounting the statement and closing it at the appropriate
    // time.
    Close() error

    // NumInput returns the number of placeholder parameters.
    //
    // If NumInput returns >= 0, the sql package will sanity check
    // argument counts from callers and return errors to the caller
    // before the statement's Exec or Query methods are called.
    //
    // NumInput may also return -1, if the driver doesn't know
    // its number of placeholders. In that case, the sql package
    // will not sanity check Exec or Query argument counts.
    NumInput() int

    // Exec executes a query that doesn't return rows, such
    // as an INSERT or UPDATE.
    Exec(args []Value) (Result, error)

    // Exec executes a query that may return rows, such as a
    // SELECT.
    Query(args []Value) (Rows, error)
}
```

Stmt is a prepared statement. It is bound to a Conn and not used by multiple goroutines concurrently.

type Tx

```
type Tx interface {  
    Commit() error  
    Rollback() error  
}
```

Tx is a transaction.

type [Value](#)

```
type Value interface{}
```

A driver Value is a value that drivers must be able to handle. A Value is either nil or an instance of one of these types:

```
int64  
float64  
bool  
[]byte  
string  [*] everywhere except from Rows.Next.  
time.Time
```

type [ValueConverter](#)

```
type ValueConverter interface {  
    // ConvertValue converts a value to a driver Value.  
    ConvertValue(v interface{}) (Value, error)  
}
```

ValueConverter is the interface providing the ConvertValue method.

Various implementations of ValueConverter are provided by the driver package to provide consistent implementations of conversions between drivers. The ValueConverters have several uses:

- * converting from the Value types as provided by the sql package into a database table's specific column type and making sure it fits, such as making sure a particular int64 fits in a table's uint16 column.
- * converting a value as given from the database into one of the driver Value types.
- * by the sql package, for converting from a driver's Value type to a user's type in a scan.

type [Valuer](#)

```
type Valuer interface {  
    // Value returns a driver Value.  
    Value() (Value, error)  
}
```

Valuer is the interface providing the Value method.

Types implementing Valuer interface are able to convert themselves to a driver Value.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg/debug

Name	Synopsis
dwarf	Package dwarf provides access to DWARF debugging information loaded from executable files, as defined in the DWARF 2.0 Standard at http://dwarfstd.org/doc/dwarf-2.0.0.pdf
elf	Package elf implements access to ELF object files.
gosym	Package gosym implements access to the Go symbol and line number tables embedded in Go binaries generated by the gc compilers.
macho	Package macho implements access to Mach-O object files.
pe	Package pe implements access to PE (Microsoft Windows Portable Executable) files.

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package dwarf

```
import "debug/dwarf"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package dwarf provides access to DWARF debugging information loaded from executable files, as defined in the DWARF 2.0 Standard at <http://dwarfstd.org/doc/dwarf-2.0.0.pdf>

Index

[type AddrType](#)

[type ArrayType](#)

[func \(t *ArrayType\) Size\(\) int64](#)

[func \(t *ArrayType\) String\(\) string](#)

[type Attr](#)

[func \(a Attr\) GoString\(\) string](#)

[func \(a Attr\) String\(\) string](#)

[type BasicType](#)

[func \(b *BasicType\) Basic\(\) *BasicType](#)

[func \(t *BasicType\) String\(\) string](#)

[type BoolType](#)

[type CharType](#)

[type CommonType](#)

[func \(c *CommonType\) Common\(\) *CommonType](#)

[func \(c *CommonType\) Size\(\) int64](#)

[type ComplexType](#)

[type Data](#)

[func New\(abbrev, aranges, frame, info, line, pubnames, ranges, str \[\]byte\) \(*Data, error\)](#)

[func \(d *Data\) Reader\(\) *Reader](#)

[func \(d *Data\) Type\(off Offset\) \(Type, error\)](#)

[type DecodeError](#)

[func \(e DecodeError\) Error\(\) string](#)

[type DotDotDotType](#)

[func \(t *DotDotDotType\) String\(\) string](#)

[type Entry](#)

[func \(e *Entry\) Val\(a Attr\) interface{}](#)

[type EnumType](#)

[func \(t *EnumType\) String\(\) string](#)

[type EnumValue](#)

[type Field](#)

[type FloatType](#)

[type FuncType](#)

[func \(t *FuncType\) String\(\) string](#)

[type IntType](#)

[type Offset](#)
[type PtrType](#)
 [func \(t *PtrType\) String\(\) string](#)
[type QualType](#)
 [func \(t *QualType\) Size\(\) int64](#)
 [func \(t *QualType\) String\(\) string](#)
[type Reader](#)
 [func \(r *Reader\) Next\(\) \(*Entry, error\)](#)
 [func \(r *Reader\) Seek\(off Offset\)](#)
 [func \(r *Reader\) SkipChildren\(\)](#)
[type StructField](#)
[type StructType](#)
 [func \(t *StructType\) Defn\(\) string](#)
 [func \(t *StructType\) String\(\) string](#)
[type Tag](#)
 [func \(t Tag\) GoString\(\) string](#)
 [func \(t Tag\) String\(\) string](#)
[type Type](#)
[type TypedefType](#)
 [func \(t *TypedefType\) Size\(\) int64](#)
 [func \(t *TypedefType\) String\(\) string](#)
[type UcharType](#)
[type UintType](#)
[type VoidType](#)
 [func \(t *VoidType\) String\(\) string](#)

Package files

[buf.go](#) [const.go](#) [entry.go](#) [open.go](#) [type.go](#) [unit.go](#)

type [AddrType](#)

```
type AddrType struct {  
    BasicType  
}
```

An AddrType represents a machine address type.

type [ArrayType](#)

```
type ArrayType struct {
    CommonType
    Type          Type
    StrideBitSize int64 // if > 0, number of bits to hold each eleme
    Count         int64 // if == -1, an incomplete array, like char
}
```

An ArrayType represents a fixed size array type.

func (***ArrayType**) [Size](#)

```
func (t *ArrayType) Size() int64
```

func (***ArrayType**) [String](#)

```
func (t *ArrayType) String() string
```

type [Attr](#)

type Attr uint32

An Attr identifies the attribute type in a DWARF Entry's Field.

```
const (
    AttrSibling           Attr = 0x01
    AttrLocation         Attr = 0x02
    AttrName             Attr = 0x03
    AttrOrdering        Attr = 0x09
    AttrByteSize        Attr = 0x0B
    AttrBitOffset       Attr = 0x0C
    AttrBitSize         Attr = 0x0D
    AttrStmtList        Attr = 0x10
    AttrLowpc           Attr = 0x11
    AttrHighpc          Attr = 0x12
    AttrLanguage        Attr = 0x13
    AttrDiscr           Attr = 0x15
    AttrDiscrValue      Attr = 0x16
    AttrVisibility       Attr = 0x17
    AttrImport          Attr = 0x18
    AttrStringLength    Attr = 0x19
    AttrCommonRef       Attr = 0x1A
    AttrCompDir         Attr = 0x1B
    AttrConstValue      Attr = 0x1C
    AttrContainingType  Attr = 0x1D
    AttrDefaultValue    Attr = 0x1E
    AttrInline          Attr = 0x20
    AttrIsOptional      Attr = 0x21
    AttrLowerBound      Attr = 0x22
    AttrProducer        Attr = 0x25
    AttrPrototyped      Attr = 0x27
    AttrReturnAddr      Attr = 0x2A
    AttrStartScope      Attr = 0x2C
    AttrStrideSize      Attr = 0x2E
    AttrUpperBound      Attr = 0x2F
    AttrAbstractOrigin  Attr = 0x31
    AttrAccessibility   Attr = 0x32
    AttrAddrClass       Attr = 0x33
    AttrArtificial       Attr = 0x34
    AttrBaseTypes       Attr = 0x35
    AttrCalling         Attr = 0x36
    AttrCount           Attr = 0x37
    AttrDataMemberLoc   Attr = 0x38
    AttrDeclColumn      Attr = 0x39
    AttrDeclFile        Attr = 0x3A
```

AttrDeclLine	Attr = 0x3B
AttrDeclaration	Attr = 0x3C
AttrDiscrList	Attr = 0x3D
AttrEncoding	Attr = 0x3E
AttrExternal	Attr = 0x3F
AttrFrameBase	Attr = 0x40
AttrFriend	Attr = 0x41
AttrIdentifierCase	Attr = 0x42
AttrMacroInfo	Attr = 0x43
AttrNameListItem	Attr = 0x44
AttrPriority	Attr = 0x45
AttrSegment	Attr = 0x46
AttrSpecification	Attr = 0x47
AttrStaticLink	Attr = 0x48
AttrType	Attr = 0x49
AttrUseLocation	Attr = 0x4A
AttrVarParam	Attr = 0x4B
AttrVirtuality	Attr = 0x4C
AttrVtableElemLoc	Attr = 0x4D
AttrAllocated	Attr = 0x4E
AttrAssociated	Attr = 0x4F
AttrDataLocation	Attr = 0x50
AttrStride	Attr = 0x51
AttrEntryPC	Attr = 0x52
AttrUseUTF8	Attr = 0x53
AttrExtension	Attr = 0x54
AttrRanges	Attr = 0x55
AttrTrampoline	Attr = 0x56
AttrCallColumn	Attr = 0x57
AttrCallFile	Attr = 0x58
AttrCallLine	Attr = 0x59
AttrDescription	Attr = 0x5A

)

func (Attr) [GoString](#)

```
func (a Attr) GoString() string
```

func (Attr) [String](#)

```
func (a Attr) String() string
```

type [BasicType](#)

```
type BasicType struct {  
    CommonType  
    BitSize    int64  
    BitOffset  int64  
}
```

A BasicType holds fields common to all basic types.

func (***BasicType**) [Basic](#)

```
func (b *BasicType) Basic() *BasicType
```

func (***BasicType**) [String](#)

```
func (t *BasicType) String() string
```

type BoolType

```
type BoolType struct {  
    BasicType  
}
```

A BoolType represents a boolean type.

type [CharType](#)

```
type CharType struct {  
    BasicType  
}
```

A CharType represents a signed character type.

type [CommonType](#)

```
type CommonType struct {  
    ByteSize int64 // size of value of this type, in bytes  
    Name      string // name that can be used to refer to type  
}
```

A CommonType holds fields common to multiple types. If a field is not known or not applicable for a given type, the zero value is used.

func (***CommonType**) [Common](#)

```
func (c *CommonType) Common() *CommonType
```

func (***CommonType**) [Size](#)

```
func (c *CommonType) Size() int64
```

type ComplexType

```
type ComplexType struct {  
    BasicType  
}
```

A ComplexType represents a complex floating point type.

type [Data](#)

```
type Data struct {  
    // contains filtered or unexported fields  
}
```

Data represents the DWARF debugging information loaded from an executable file (for example, an ELF or Mach-O executable).

func [New](#)

```
func New(abbrev, aranges, frame, info, line, pubnames, ranges, str []
```

New returns a new Data object initialized from the given parameters. Rather than calling this function directly, clients should typically use the DWARF method of the File type of the appropriate package debug/elf, debug/macho, or debug/pe.

The []byte arguments are the data from the corresponding debug section in the object file; for example, for an ELF object, abbrev is the contents of the ".debug_abbrev" section.

func (*Data) [Reader](#)

```
func (d *Data) Reader() *Reader
```

Reader returns a new Reader for Data. The reader is positioned at byte offset 0 in the DWARF “info” section.

func (*Data) [Type](#)

```
func (d *Data) Type(off Offset) (Type, error)
```

type [DecodeError](#)

```
type DecodeError struct {  
    Name    string  
    Offset  Offset  
    Err     string  
}
```

func (DecodeError) [Error](#)

```
func (e DecodeError) Error() string
```

type [DotDotDotType](#)

```
type DotDotDotType struct {  
    CommonType  
}
```

A DotDotDotType represents the variadic ... function parameter.

func (*DotDotDotType) [String](#)

```
func (t *DotDotDotType) String() string
```

type [Entry](#)

```
type Entry struct {
    Offset    Offset // offset of Entry in DWARF info
    Tag       Tag      // tag (kind of Entry)
    Children  bool    // whether Entry is followed by children
    Field     []Field
}
```

An entry is a sequence of attribute/value pairs.

func (*Entry) [Val](#)

```
func (e *Entry) Val(a Attr) interface{}
```

Val returns the value associated with attribute Attr in Entry, or nil if there is no such attribute.

A common idiom is to merge the check for nil return with the check that the value has the expected dynamic type, as in:

```
v, ok := e.Val(AttrSibling).(int64);
```

type [EnumType](#)

```
type EnumType struct {  
    CommonType  
    EnumName string  
    Val      []*EnumValue  
}
```

An EnumType represents an enumerated type. The only indication of its native integer type is its ByteSize (inside CommonType).

func (*EnumType) [String](#)

```
func (t *EnumType) String() string
```

type EnumValue

```
type EnumValue struct {  
    Name string  
    Val  int64  
}
```

An EnumValue represents a single enumeration value.

type Field

```
type Field struct {  
    Attr Attr  
    Val  interface{  
}
```

A Field is a single attribute/value pair in an Entry.

type [FloatType](#)

```
type FloatType struct {  
    BasicType  
}
```

A FloatType represents a floating point type.

type [FuncType](#)

```
type FuncType struct {  
    CommonType  
    Returntype Type  
    ParamType []Type  
}
```

A FuncType represents a function type.

func (*FuncType) [String](#)

```
func (t *FuncType) String() string
```

type [IntType](#)

```
type IntType struct {  
    BasicType  
}
```

An IntType represents a signed integer type.

type [Offset](#)

```
type Offset uint32
```

An Offset represents the location of an Entry within the DWARF info. (See Reader.Seek.)

type [PtrType](#)

```
type PtrType struct {  
    CommonType  
    Type Type  
}
```

A PtrType represents a pointer type.

func (*PtrType) [String](#)

```
func (t *PtrType) String() string
```

type [QualType](#)

```
type QualType struct {  
    CommonType  
    Qual string  
    Type Type  
}
```

A QualType represents a type that has the C/C++ "const", "restrict", or "volatile" qualifier.

func (***QualType**) [Size](#)

```
func (t *QualType) Size() int64
```

func (***QualType**) [String](#)

```
func (t *QualType) String() string
```

type [Reader](#)

```
type Reader struct {  
    // contains filtered or unexported fields  
}
```

A Reader allows reading Entry structures from a DWARF “info” section. The Entry structures are arranged in a tree. The Reader's Next function return successive entries from a pre-order traversal of the tree. If an entry has children, its Children field will be true, and the children follow, terminated by an Entry with Tag 0.

func (*Reader) [Next](#)

```
func (r *Reader) Next() (*Entry, error)
```

Next reads the next entry from the encoded entry stream. It returns nil, nil when it reaches the end of the section. It returns an error if the current offset is invalid or the data at the offset cannot be decoded as a valid Entry.

func (*Reader) [Seek](#)

```
func (r *Reader) Seek(off Offset)
```

Seek positions the Reader at offset off in the encoded entry stream. Offset 0 can be used to denote the first entry.

func (*Reader) [SkipChildren](#)

```
func (r *Reader) SkipChildren()
```

SkipChildren skips over the child entries associated with the last Entry returned by Next. If that Entry did not have children or Next has not been called, SkipChildren is a no-op.

type [StructField](#)

```
type StructField struct {
    Name      string
    Type      Type
    ByteOffset int64
    ByteSize  int64
    BitOffset  int64 // within the ByteSize bytes at ByteOffset
    BitSize   int64 // zero if not a bit field
}
```

A StructField represents a field in a struct, union, or C++ class type.

type [StructType](#)

```
type StructType struct {
    CommonType
    StructName string
    Kind        string // "struct", "union", or "class".
    Field       []*StructField
    Incomplete bool // if true, struct, union, class is declared but
}
```

A StructType represents a struct, union, or C++ class type.

func (*StructType) [Defn](#)

```
func (t *StructType) Defn() string
```

func (*StructType) [String](#)

```
func (t *StructType) String() string
```

type [Tag](#)

```
type Tag uint32
```

A Tag is the classification (the type) of an Entry.

```
const (  
    TagArrayType           Tag = 0x01  
    TagClassType          Tag = 0x02  
    TagEntryPoint         Tag = 0x03  
    TagEnumerationType    Tag = 0x04  
    TagFormalParameter    Tag = 0x05  
    TagImportedDeclaration Tag = 0x08  
    TagLabel              Tag = 0x0A  
    TagLexDwarfBlock      Tag = 0x0B  
    TagMember             Tag = 0x0D  
    TagPointerType        Tag = 0x0F  
    TagReferenceType      Tag = 0x10  
    TagCompileUnit        Tag = 0x11  
    TagStringType         Tag = 0x12  
    TagStructType         Tag = 0x13  
    TagSubroutineType     Tag = 0x15  
    TagTypedef            Tag = 0x16  
    TagUnionType          Tag = 0x17  
    TagUnspecifiedParameters Tag = 0x18  
    TagVariant            Tag = 0x19  
    TagCommonDwarfBlock   Tag = 0x1A  
    TagCommonInclusion      Tag = 0x1B  
    TagInheritance        Tag = 0x1C  
    TagInlinedSubroutine  Tag = 0x1D  
    TagModule             Tag = 0x1E  
    TagPtrToMemberType    Tag = 0x1F  
    TagSetType            Tag = 0x20  
    TagSubrangeType       Tag = 0x21  
    TagWithStmt           Tag = 0x22  
    TagAccessDeclaration  Tag = 0x23  
    TagBaseType           Tag = 0x24  
    TagCatchDwarfBlock    Tag = 0x25  
    TagConstType          Tag = 0x26  
    TagConstant           Tag = 0x27  
    TagEnumerator         Tag = 0x28  
    TagFileType           Tag = 0x29  
    TagFriend             Tag = 0x2A  
    TagNameList           Tag = 0x2B  
    TagNameListItem       Tag = 0x2C  
    TagPackedType         Tag = 0x2D  
    TagSubprogram         Tag = 0x2E
```

TagTemplateTypeParameter	Tag = 0x2F
TagTemplateValueParameter	Tag = 0x30
TagThrownType	Tag = 0x31
TagTryDwarfBlock	Tag = 0x32
TagVariantPart	Tag = 0x33
TagVariable	Tag = 0x34
TagVolatileType	Tag = 0x35
TagDwarfProcedure	Tag = 0x36
TagRestrictType	Tag = 0x37
TagInterfaceType	Tag = 0x38
TagNamespace	Tag = 0x39
TagImportedModule	Tag = 0x3A
TagUnspecifiedType	Tag = 0x3B
TagPartialUnit	Tag = 0x3C
TagImportedUnit	Tag = 0x3D
TagMutableType	Tag = 0x3E

)

func (Tag) [GoString](#)

```
func (t Tag) GoString() string
```

func (Tag) [String](#)

```
func (t Tag) String() string
```

type Type

```
type Type interface {  
    Common() *CommonType  
    String() string  
    Size() int64  
}
```

A Type conventionally represents a pointer to any of the specific Type structures (CharType, StructType, etc.).

type [TypedefType](#)

```
type TypedefType struct {  
    CommonType  
    Type Type  
}
```

A TypedefType represents a named type.

func (*TypedefType) [Size](#)

```
func (t *TypedefType) Size() int64
```

func (*TypedefType) [String](#)

```
func (t *TypedefType) String() string
```

type [UcharType](#)

```
type UcharType struct {  
    BasicType  
}
```

A UcharType represents an unsigned character type.

type [UIntType](#)

```
type UIntType struct {  
    BasicType  
}
```

A UIntType represents an unsigned integer type.

type [VoidType](#)

```
type VoidType struct {  
    CommonType  
}
```

A VoidType represents the C void type.

func (*VoidType) [String](#)

```
func (t *VoidType) String() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package elf

```
import "debug/elf"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package elf implements access to ELF object files.

Index

Constants

[func R_INFO\(sym, typ uint32\) uint64](#)

[func R_INFO32\(sym, typ uint32\) uint32](#)

[func R_SYM32\(info uint32\) uint32](#)

[func R_SYM64\(info uint64\) uint32](#)

[func R_TYPE32\(info uint32\) uint32](#)

[func R_TYPE64\(info uint64\) uint32](#)

[func ST_INFO\(bind SymBind, typ SymType\) uint8](#)

type Class

[func \(i Class\) GoString\(\) string](#)

[func \(i Class\) String\(\) string](#)

type Data

[func \(i Data\) GoString\(\) string](#)

[func \(i Data\) String\(\) string](#)

type Dyn32

type Dyn64

type DynFlag

[func \(i DynFlag\) GoString\(\) string](#)

[func \(i DynFlag\) String\(\) string](#)

type DynTag

[func \(i DynTag\) GoString\(\) string](#)

[func \(i DynTag\) String\(\) string](#)

type File

[func NewFile\(r io.ReaderAt\) \(*File, error\)](#)

[func Open\(name string\) \(*File, error\)](#)

[func \(f *File\) Close\(\) error](#)

[func \(f *File\) DWARF\(\) \(*dwarf.Data, error\)](#)

[func \(f *File\) ImportedLibraries\(\) \(\[\]string, error\)](#)

[func \(f *File\) ImportedSymbols\(\) \(\[\]ImportedSymbol, error\)](#)

[func \(f *File\) Section\(name string\) *Section](#)

[func \(f *File\) SectionByType\(typ SectionType\) *Section](#)

[func \(f *File\) Symbols\(\) \(\[\]Symbol, error\)](#)

type FileHeader

type FormatError

[func \(e *FormatError\) Error\(\) string](#)

[type Header32](#)
[type Header64](#)
[type ImportedSymbol](#)
[type Machine](#)
 [func \(i Machine\) GoString\(\) string](#)
 [func \(i Machine\) String\(\) string](#)
[type NType](#)
 [func \(i NType\) GoString\(\) string](#)
 [func \(i NType\) String\(\) string](#)
[type OSABI](#)
 [func \(i OSABI\) GoString\(\) string](#)
 [func \(i OSABI\) String\(\) string](#)
[type Prog](#)
 [func \(p *Prog\) Open\(\) io.ReadSeeker](#)
[type Prog32](#)
[type Prog64](#)
[type ProgFlag](#)
 [func \(i ProgFlag\) GoString\(\) string](#)
 [func \(i ProgFlag\) String\(\) string](#)
[type ProgHeader](#)
[type ProgType](#)
 [func \(i ProgType\) GoString\(\) string](#)
 [func \(i ProgType\) String\(\) string](#)
[type R_386](#)
 [func \(i R_386\) GoString\(\) string](#)
 [func \(i R_386\) String\(\) string](#)
[type R_ALPHA](#)
 [func \(i R_ALPHA\) GoString\(\) string](#)
 [func \(i R_ALPHA\) String\(\) string](#)
[type R_ARM](#)
 [func \(i R_ARM\) GoString\(\) string](#)
 [func \(i R_ARM\) String\(\) string](#)
[type R_PPC](#)
 [func \(i R_PPC\) GoString\(\) string](#)
 [func \(i R_PPC\) String\(\) string](#)
[type R_SPARC](#)
 [func \(i R_SPARC\) GoString\(\) string](#)
 [func \(i R_SPARC\) String\(\) string](#)
[type R_X86_64](#)

[func \(i R_X86_64\) GoString\(\) string](#)

[func \(i R_X86_64\) String\(\) string](#)

[type Rel32](#)

[type Rel64](#)

[type Rela32](#)

[type Rela64](#)

[type Section](#)

[func \(s *Section\) Data\(\) \(\[\]byte, error\)](#)

[func \(s *Section\) Open\(\) io.ReadSeeker](#)

[type Section32](#)

[type Section64](#)

[type SectionFlag](#)

[func \(i SectionFlag\) GoString\(\) string](#)

[func \(i SectionFlag\) String\(\) string](#)

[type SectionHeader](#)

[type SectionIndex](#)

[func \(i SectionIndex\) GoString\(\) string](#)

[func \(i SectionIndex\) String\(\) string](#)

[type SectionType](#)

[func \(i SectionType\) GoString\(\) string](#)

[func \(i SectionType\) String\(\) string](#)

[type Sym32](#)

[type Sym64](#)

[type SymBind](#)

[func ST_BIND\(info uint8\) SymBind](#)

[func \(i SymBind\) GoString\(\) string](#)

[func \(i SymBind\) String\(\) string](#)

[type SymType](#)

[func ST_TYPE\(info uint8\) SymType](#)

[func \(i SymType\) GoString\(\) string](#)

[func \(i SymType\) String\(\) string](#)

[type SymVis](#)

[func ST_VISIBILITY\(other uint8\) SymVis](#)

[func \(i SymVis\) GoString\(\) string](#)

[func \(i SymVis\) String\(\) string](#)

[type Symbol](#)

[type Type](#)

[func \(i Type\) GoString\(\) string](#)

[func \(i Type\) String\(\) string](#)

[type Version](#)
[func \(i Version\) GoString\(\) string](#)
[func \(i Version\) String\(\) string](#)

Package files

[elf.go](#) [file.go](#)

Constants

```
const (  
    EI_CLASS      = 4 /* Class of machine. */  
    EI_DATA       = 5 /* Data format. */  
    EI_VERSION    = 6 /* ELF format version. */  
    EI_OSABI      = 7 /* Operating system / ABI identification */  
    EI_ABIVERSION = 8 /* ABI version */  
    EI_PAD        = 9 /* Start of padding (per SVR4 ABI). */  
    EI_NIDENT     = 16 /* Size of e_ident array. */  
)
```

Indexes into the Header.Ident array.

```
const ARM_MAGIC_TRAMP_NUMBER = 0x5c000003
```

Magic number for the elf trampoline, chosen wisely to be an immediate value.

```
const ELFMAG = "\177ELF"
```

Initial magic number for ELF files.

```
const Sym32Size = 16
```

```
const Sym64Size = 24
```

func [R_INFO](#)

```
func R_INFO(sym, typ uint32) uint64
```

func [R_INFO32](#)

func R_INFO32(sym, typ uint32) uint32

func [R_SYM32](#)

```
func R_SYM32(info uint32) uint32
```

func [R_SYM64](#)

func R_SYM64(info uint64) uint32

func R_TYPE32

```
func R_TYPE32(info uint32) uint32
```

func R_TYPE64

```
func R_TYPE64(info uint64) uint32
```

func [ST_INFO](#)

```
func ST_INFO(bind SymBind, typ SymType) uint8
```

type [Class](#)

type Class byte

Class is found in Header.Ident[EI_CLASS] and Header.Class.

```
const (  
    ELFCLASSNONE Class = 0 /* Unknown class. */  
    ELFCLASS32   Class = 1 /* 32-bit architecture. */  
    ELFCLASS64   Class = 2 /* 64-bit architecture. */  
)
```

func (Class) [GoString](#)

func (i Class) GoString() string

func (Class) [String](#)

func (i Class) String() string

type [Data](#)

type Data byte

Data is found in Header.Ident[EI_DATA] and Header.Data.

```
const (  
    ELFDATANONE Data = 0 /* Unknown data format. */  
    ELFDATA2LSB Data = 1 /* 2's complement little-endian. */  
    ELFDATA2MSB Data = 2 /* 2's complement big-endian. */  
)
```

func (Data) [GoString](#)

func (i Data) GoString() string

func (Data) [String](#)

func (i Data) String() string

type [Dyn32](#)

```
type Dyn32 struct {
    Tag int32 /* Entry type. */
    Val uint32 /* Integer/Address value. */
}
```

ELF32 Dynamic structure. The ".dynamic" section contains an array of them.

type [Dyn64](#)

```
type Dyn64 struct {  
    Tag int64 /* Entry type. */  
    Val uint64 /* Integer/address value */  
}
```

ELF64 Dynamic structure. The ".dynamic" section contains an array of them.

type [DynFlag](#)

```
type DynFlag int
```

DT_FLAGS values.

```
const (  
    DF_ORIGIN DynFlag = 0x0001 /* Indicates that the object being lo  
        make reference to the  
        $ORIGIN substitution string */  
    DF_SYMBOLIC DynFlag = 0x0002 /* Indicates "symbolic" linking. */  
    DF_TEXTREL DynFlag = 0x0004 /* Indicates there may be relocatio  
    DF_BIND_NOW DynFlag = 0x0008 /* Indicates that the dynamic linke  
        process all relocations for the object  
        containing this entry before transferring  
        control to the program. */  
    DF_STATIC_TLS DynFlag = 0x0010 /* Indicates that the shared obje  
        executable contains code using a static  
        thread-local storage scheme. */  
)
```

func (DynFlag) [GoString](#)

```
func (i DynFlag) GoString() string
```

func (DynFlag) [String](#)

```
func (i DynFlag) String() string
```

type [DynTag](#)

```
type DynTag int
```

```
Dyn.Tag
```

```
const (
    DT_NULL          DynTag = 0 /* Terminating entry. */
    DT_NEEDED        DynTag = 1 /* String table offset of a needed s
    DT_PLTRELSZ      DynTag = 2 /* Total size in bytes of PLT reloca
    DT_PLTGOT        DynTag = 3 /* Processor-dependent address. */
    DT_HASH          DynTag = 4 /* Address of symbol hash table. */
    DT_STRTAB        DynTag = 5 /* Address of string table. */
    DT_SYMTAB        DynTag = 6 /* Address of symbol table. */
    DT_RELA          DynTag = 7 /* Address of ElfNN_Rela relocations
    DT_RELASZ        DynTag = 8 /* Total size of ElfNN_Rela relocati
    DT_RELAENT       DynTag = 9 /* Size of each ElfNN_Rela relocatio
    DT_STRSZ         DynTag = 10 /* Size of string table. */
    DT_SYMENT        DynTag = 11 /* Size of each symbol table entry.
    DT_INIT          DynTag = 12 /* Address of initialization functio
    DT_FINI          DynTag = 13 /* Address of finalization function.
    DT_SONAME        DynTag = 14 /* String table offset of shared obj
    DT_RPATH         DynTag = 15 /* String table offset of library pa
    DT_SYMBOLIC      DynTag = 16 /* Indicates "symbolic" linking. [su
    DT_REL           DynTag = 17 /* Address of ElfNN_Rel relocations.
    DT_RELSZ         DynTag = 18 /* Total size of ElfNN_Rel relocatio
    DT_RELENT        DynTag = 19 /* Size of each ElfNN_Rel relocation
    DT_PLTREL        DynTag = 20 /* Type of relocation used for PLT.
    DT_DEBUG         DynTag = 21 /* Reserved (not used). */
    DT_TEXTREL       DynTag = 22 /* Indicates there may be relocation
    DT_JMPREL        DynTag = 23 /* Address of PLT relocations. */
    DT_BIND_NOW      DynTag = 24 /* [sup] */
    DT_INIT_ARRAY    DynTag = 25 /* Address of the array of pointers
    DT_FINI_ARRAY    DynTag = 26 /* Address of the array of pointers
    DT_INIT_ARRAYSZ  DynTag = 27 /* Size in bytes of the array of ini
    DT_FINI_ARRAYSZ  DynTag = 28 /* Size in bytes of the array of ter
    DT_RUNPATH       DynTag = 29 /* String table offset of a null-ter
    DT_FLAGS         DynTag = 30 /* Object specific flag values. */
    DT_ENCODING      DynTag = 32 /* Values greater than or equal to D
    and less than DT_LOOS follow the rules for
    the interpretation of the d_un union
    as follows: even == 'd_ptr', even == 'd_val'
    or none */
    DT_PREINIT_ARRAY DynTag = 32 /* Address of the array o
    DT_PREINIT_ARRAYSZ DynTag = 33 /* Size in bytes of the a
    DT_LOOS           DynTag = 0x6000000d /* First OS-specific */
    DT_HIOS           DynTag = 0x6ffff000 /* Last OS-specific */
```

```
DT_VERSYM          DynTag = 0x6fffffff0
DT_VERNEED         DynTag = 0x6fffffffef
DT_VERNEEDNUM      DynTag = 0x6fffffffef
DT_LOPROC          DynTag = 0x70000000 /* First processor-specific
DT_HIPROC          DynTag = 0x7fffffff /* Last processor-specific
)
```

func (DynTag) [GoString](#)

```
func (i DynTag) GoString() string
```

func (DynTag) [String](#)

```
func (i DynTag) String() string
```

type [File](#)

```
type File struct {
    FileHeader
    Sections []*Section
    Progs     []*Prog
    // contains filtered or unexported fields
}
```

A File represents an open ELF file.

func [NewFile](#)

```
func NewFile(r io.ReaderAt) (*File, error)
```

NewFile creates a new File for accessing an ELF binary in an underlying reader. The ELF binary is expected to start at position 0 in the ReaderAt.

func [Open](#)

```
func Open(name string) (*File, error)
```

Open opens the named file using os.Open and prepares it for use as an ELF binary.

func (*File) [Close](#)

```
func (f *File) Close() error
```

Close closes the File. If the File was created using NewFile directly instead of Open, Close has no effect.

func (*File) [DWARF](#)

```
func (f *File) DWARF() (*dwarf.Data, error)
```

func (*File) [ImportedLibraries](#)

```
func (f *File) ImportedLibraries() ([]string, error)
```

ImportedLibraries returns the names of all libraries referred to by the binary f that are expected to be linked with the binary at dynamic link time.

func (*File) [ImportedSymbols](#)

```
func (f *File) ImportedSymbols() ([]ImportedSymbol, error)
```

ImportedSymbols returns the names of all symbols referred to by the binary f that are expected to be satisfied by other libraries at dynamic load time. It does not return weak symbols.

func (*File) [Section](#)

```
func (f *File) Section(name string) *Section
```

Section returns a section with the given name, or nil if no such section exists.

func (*File) [SectionByType](#)

```
func (f *File) SectionByType(typ SectionType) *Section
```

SectionByType returns the first section in f with the given type, or nil if there is no such section.

func (*File) [Symbols](#)

```
func (f *File) Symbols() ([]Symbol, error)
```

Symbols returns the symbol table for f.

type [FileHeader](#)

```
type FileHeader struct {  
    Class      Class  
    Data       Data  
    Version    Version  
    OSABI      OSABI  
    ABIVersion uint8  
    ByteOrder  binary.ByteOrder  
    Type       Type  
    Machine    Machine  
}
```

A FileHeader represents an ELF file header.

type [FormatError](#)

```
type FormatError struct {  
    // contains filtered or unexported fields  
}
```

func ([*FormatError](#)) [Error](#)

```
func (e *FormatError) Error() string
```

type Header32

```
type Header32 struct {
    Ident    [EI_NIDENT]byte /* File identification. */
    Type     uint16      /* File type. */
    Machine  uint16      /* Machine architecture. */
    Version  uint32      /* ELF format version. */
    Entry    uint32      /* Entry point. */
    Phoff    uint32      /* Program header file offset. */
    Shoff    uint32      /* Section header file offset. */
    Flags    uint32      /* Architecture-specific flags. */
    Ehsize   uint16      /* Size of ELF header in bytes. */
    Phentsize uint16     /* Size of program header entry. */
    Phnum    uint16     /* Number of program header entries. */
    Shentsize uint16     /* Size of section header entry. */
    Shnum    uint16     /* Number of section header entries. */
    Shstrndx uint16     /* Section name strings section. */
}
```

ELF32 File header.

type Header64

```
type Header64 struct {
    Ident    [EI_NIDENT]byte /* File identification. */
    Type     uint16        /* File type. */
    Machine  uint16        /* Machine architecture. */
    Version  uint32        /* ELF format version. */
    Entry    uint64        /* Entry point. */
    Phoff    uint64        /* Program header file offset. */
    Shoff    uint64        /* Section header file offset. */
    Flags    uint32        /* Architecture-specific flags. */
    Ehsize   uint16        /* Size of ELF header in bytes. */
    Phentsize uint16       /* Size of program header entry. */
    Phnum    uint16        /* Number of program header entries. *
    Shentsize uint16       /* Size of section header entry. */
    Shnum    uint16        /* Number of section header entries. *
    Shstrndx uint16        /* Section name strings section. */
}
```

ELF64 file header.

type ImportedSymbol

```
type ImportedSymbol struct {  
    Name    string  
    Version string  
    Library string  
}
```

type [Machine](#)

type Machine uint16

Machine is found in Header.Machine.

```
const (
    EM_NONE      Machine = 0 /* Unknown machine. */
    EM_M32       Machine = 1 /* AT&T WE32100. */
    EM_SPARC     Machine = 2 /* Sun SPARC. */
    EM_386       Machine = 3 /* Intel i386. */
    EM_68K       Machine = 4 /* Motorola 68000. */
    EM_88K       Machine = 5 /* Motorola 88000. */
    EM_860       Machine = 7 /* Intel i860. */
    EM_MIPS      Machine = 8 /* MIPS R3000 Big-Endian only. */
    EM_S370      Machine = 9 /* IBM System/370. */
    EM_MIPS_RS3_LE Machine = 10 /* MIPS R3000 Little-Endian. */
    EM_PARISC    Machine = 15 /* HP PA-RISC. */
    EM_VPP500    Machine = 17 /* Fujitsu VPP500. */
    EM_SPARC32PLUS Machine = 18 /* SPARC v8plus. */
    EM_960       Machine = 19 /* Intel 80960. */
    EM_PPC       Machine = 20 /* PowerPC 32-bit. */
    EM_PPC64     Machine = 21 /* PowerPC 64-bit. */
    EM_S390      Machine = 22 /* IBM System/390. */
    EM_V800      Machine = 36 /* NEC V800. */
    EM_FR20      Machine = 37 /* Fujitsu FR20. */
    EM_RH32      Machine = 38 /* TRW RH-32. */
    EM_RCE       Machine = 39 /* Motorola RCE. */
    EM_ARM       Machine = 40 /* ARM. */
    EM_SH        Machine = 42 /* Hitachi SH. */
    EM_SPARCV9   Machine = 43 /* SPARC v9 64-bit. */
    EM_TRICORE   Machine = 44 /* Siemens TriCore embedded processoro
    EM_ARC       Machine = 45 /* Argonaut RISC Core. */
    EM_H8_300    Machine = 46 /* Hitachi H8/300. */
    EM_H8_300H   Machine = 47 /* Hitachi H8/300H. */
    EM_H8S       Machine = 48 /* Hitachi H8S. */
    EM_H8_500    Machine = 49 /* Hitachi H8/500. */
    EM_IA_64     Machine = 50 /* Intel IA-64 Processor. */
    EM_MIPS_X    Machine = 51 /* Stanford MIPS-X. */
    EM_COLDFIRE  Machine = 52 /* Motorola ColdFire. */
    EM_68HC12    Machine = 53 /* Motorola M68HC12. */
    EM_MMA       Machine = 54 /* Fujitsu MMA. */
    EM_PCP       Machine = 55 /* Siemens PCP. */
    EM_NCPU      Machine = 56 /* Sony nCPU. */
    EM_NDR1      Machine = 57 /* Denso NDR1 microprocessor. */
    EM_STARCORE  Machine = 58 /* Motorola Star*Core processor. */
    EM_ME16      Machine = 59 /* Toyota ME16 processor. */
```

```
EM_ST100      Machine = 60 /* STMicroelectronics ST100 process  
EM_TINYJ      Machine = 61 /* Advanced Logic Corp. TinyJ proces  
EM_X86_64     Machine = 62
```

```
/* Non-standard or deprecated. */
```

```
EM_486        Machine = 6      /* Intel i486. */  
EM_MIPS_RS4_BE Machine = 10     /* MIPS R4000 Big-Endian */  
EM_ALPHA_STD  Machine = 41     /* Digital Alpha (standard value  
EM_ALPHA      Machine = 0x9026 /* Alpha (written in the absence
```

```
)
```

func (Machine) [GoString](#)

```
func (i Machine) GoString() string
```

func (Machine) [String](#)

```
func (i Machine) String() string
```

type [NType](#)

```
type NType int
```

NType values; used in core files.

```
const (  
    NT_PRSTATUS NType = 1 /* Process status. */  
    NT_FPREGSET NType = 2 /* Floating point registers. */  
    NT_PRPSINFO NType = 3 /* Process state info. */  
)
```

func (NType) [GoString](#)

```
func (i NType) GoString() string
```

func (NType) [String](#)

```
func (i NType) String() string
```

type [OSABI](#)

type OSABI byte

OSABI is found in Header.Ident[EI_OSABI] and Header.OSABI.

```
const (
    ELFOSABI_NONE           OSABI = 0   /* UNIX System V ABI */
    ELFOSABI_HPUX           OSABI = 1   /* HP-UX operating system */
    ELFOSABI_NETBSD         OSABI = 2   /* NetBSD */
    ELFOSABI_LINUX          OSABI = 3   /* GNU/Linux */
    ELFOSABI_HURD           OSABI = 4   /* GNU/Hurd */
    ELFOSABI_86OPEN         OSABI = 5   /* 86Open common IA32 ABI */
    ELFOSABI_SOLARIS        OSABI = 6   /* Solaris */
    ELFOSABI_AIX            OSABI = 7   /* AIX */
    ELFOSABI_IRIX           OSABI = 8   /* IRIX */
    ELFOSABI_FREEBSD        OSABI = 9   /* FreeBSD */
    ELFOSABI_TRU64          OSABI = 10  /* TRU64 UNIX */
    ELFOSABI_MODESTO        OSABI = 11  /* Novell Modesto */
    ELFOSABI_OPENBSD        OSABI = 12  /* OpenBSD */
    ELFOSABI_OPENVMS        OSABI = 13  /* Open VMS */
    ELFOSABI_NSK            OSABI = 14  /* HP Non-Stop Kernel */
    ELFOSABI_ARM            OSABI = 97  /* ARM */
    ELFOSABI_STANDALONE     OSABI = 255 /* Standalone (embedded) applica
)
```

func (OSABI) [GoString](#)

func (i OSABI) GoString() string

func (OSABI) [String](#)

func (i OSABI) String() string

type [Prog](#)

```
type Prog struct {
    ProgHeader

    // Embed ReaderAt for ReadAt method.
    // Do not embed SectionReader directly
    // to avoid having Read and Seek.
    // If a client wants Read and Seek it must use
    // Open() to avoid fighting over the seek offset
    // with other clients.
    io.ReaderAt
    // contains filtered or unexported fields
}
```

A Prog represents a single ELF program header in an ELF binary.

func ([*Prog](#)) [Open](#)

```
func (p *Prog) Open() io.ReadSeeker
```

Open returns a new ReadSeeker reading the ELF program body.

type [Prog32](#)

```
type Prog32 struct {
    Type    uint32 /* Entry type. */
    Off     uint32 /* File offset of contents. */
    Vaddr   uint32 /* Virtual address in memory image. */
    Paddr   uint32 /* Physical address (not used). */
    Filesz  uint32 /* Size of contents in file. */
    Memsz   uint32 /* Size of contents in memory. */
    Flags   uint32 /* Access permission flags. */
    Align   uint32 /* Alignment in memory and file. */
}
```

ELF32 Program header.

type [Prog64](#)

```
type Prog64 struct {
    Type    uint32 /* Entry type. */
    Flags   uint32 /* Access permission flags. */
    Off     uint64 /* File offset of contents. */
    Vaddr   uint64 /* Virtual address in memory image. */
    Paddr   uint64 /* Physical address (not used). */
    Filesz  uint64 /* Size of contents in file. */
    Memsz   uint64 /* Size of contents in memory. */
    Align   uint64 /* Alignment in memory and file. */
}
```

ELF64 Program header.

type ProgFlag

```
type ProgFlag uint32
```

Prog.Flag

```
const (  
    PF_X      ProgFlag = 0x1      /* Executable. */  
    PF_W      ProgFlag = 0x2      /* Writable. */  
    PF_R      ProgFlag = 0x4      /* Readable. */  
    PF_MASKOS ProgFlag = 0x0ff00000 /* Operating system-specific.  
    PF_MASKPROC ProgFlag = 0xf0000000 /* Processor-specific. */  
)
```

func (ProgFlag) GoString

```
func (i ProgFlag) GoString() string
```

func (ProgFlag) String

```
func (i ProgFlag) String() string
```

type ProgHeader

```
type ProgHeader struct {
    Type    ProgType
    Flags   ProgFlag
    Off     uint64
    Vaddr   uint64
    Paddr   uint64
    Filesz  uint64
    Memsz   uint64
    Align   uint64
}
```

A ProgHeader represents a single ELF program header.

type [ProgType](#)

```
type ProgType int
```

```
Prog.Type
```

```
const (  
    PT_NULL      ProgType = 0      /* Unused entry. */  
    PT_LOAD      ProgType = 1      /* Loadable segment. */  
    PT_DYNAMIC   ProgType = 2      /* Dynamic linking information  
    PT_INTERP    ProgType = 3      /* Pathname of interpreter. */  
    PT_NOTE      ProgType = 4      /* Auxiliary information. */  
    PT_SHLIB     ProgType = 5      /* Reserved (not used). */  
    PT_PHDR      ProgType = 6      /* Location of program header i  
    PT_TLS       ProgType = 7      /* Thread local storage segment  
    PT_LOOS      ProgType = 0x60000000 /* First OS-specific. */  
    PT_HIOS      ProgType = 0x6fffffff /* Last OS-specific. */  
    PT_LOPROC    ProgType = 0x70000000 /* First processor-specific typ  
    PT_HIPROC    ProgType = 0x7fffffff /* Last processor-specific type  
)
```

func (ProgType) [GoString](#)

```
func (i ProgType) GoString() string
```

func (ProgType) [String](#)

```
func (i ProgType) String() string
```

type [R_386](#)

```
type R_386 int
```

Relocation types for 386.

```
const (
    R_386_NONE          R_386 = 0 /* No relocation. */
    R_386_32            R_386 = 1 /* Add symbol value. */
    R_386_PC32         R_386 = 2 /* Add PC-relative symbol value. */
    R_386_GOT32        R_386 = 3 /* Add PC-relative GOT offset. */
    R_386_PLT32        R_386 = 4 /* Add PC-relative PLT offset. */
    R_386_COPY         R_386 = 5 /* Copy data from shared object. */
    R_386_GLOB_DAT     R_386 = 6 /* Set GOT entry to data address. */
    R_386_JMP_SLOT     R_386 = 7 /* Set GOT entry to code address. */
    R_386_RELATIVE     R_386 = 8 /* Add load address of shared object. */
    R_386_GOTOFF       R_386 = 9 /* Add GOT-relative symbol address. */
    R_386_GOTPC        R_386 = 10 /* Add PC-relative GOT table address. */
    R_386_TLS_TPOFF    R_386 = 14 /* Negative offset in static TLS block. */
    R_386_TLS_IE       R_386 = 15 /* Absolute address of GOT for negative static TLS index. */
    R_386_TLS_GOTIE    R_386 = 16 /* GOT entry for negative static TLS index. */
    R_386_TLS_LE       R_386 = 17 /* Negative offset relative to static TLS index. */
    R_386_TLS_GD       R_386 = 18 /* 32 bit offset to GOT (index, offset). */
    R_386_TLS_LDM      R_386 = 19 /* 32 bit offset to GOT (index, zero). */
    R_386_TLS_GD_32    R_386 = 24 /* 32 bit offset to GOT (index, offset). */
    R_386_TLS_GD_PUSH  R_386 = 25 /* pushl instruction for Sun ABI GOT. */
    R_386_TLS_GD_CALL  R_386 = 26 /* call instruction for Sun ABI GOT. */
    R_386_TLS_GD_POP   R_386 = 27 /* popl instruction for Sun ABI GOT. */
    R_386_TLS_LDM_32   R_386 = 28 /* 32 bit offset to GOT (index, zero). */
    R_386_TLS_LDM_PUSH R_386 = 29 /* pushl instruction for Sun ABI LDM. */
    R_386_TLS_LDM_CALL R_386 = 30 /* call instruction for Sun ABI LDM. */
    R_386_TLS_LDM_POP  R_386 = 31 /* popl instruction for Sun ABI LDM. */
    R_386_TLS_LDO_32   R_386 = 32 /* 32 bit offset from start of TLS block. */
    R_386_TLS_IE_32    R_386 = 33 /* 32 bit offset to GOT static TLS index. */
    R_386_TLS_LE_32    R_386 = 34 /* 32 bit offset within static TLS block. */
    R_386_TLS_DTPMOD32 R_386 = 35 /* GOT entry containing TLS index. */
    R_386_TLS_DTPOFF32 R_386 = 36 /* GOT entry containing TLS offset. */
    R_386_TLS_TPOFF32  R_386 = 37 /* GOT entry of -ve static TLS offset. */
)
```

func (R_386) [GoString](#)

```
func (i R_386) GoString() string
```

func (R_386) [String](#)

```
func (i R_386) String() string
```

type [R_ALPHA](#)

```
type R_ALPHA int
```

Relocation types for Alpha.

```
const (
    R_ALPHA_NONE           R_ALPHA = 0 /* No reloc */
    R_ALPHA_REFLONG        R_ALPHA = 1 /* Direct 32 bit */
    R_ALPHA_REFQUAD        R_ALPHA = 2 /* Direct 64 bit */
    R_ALPHA_GPREL32        R_ALPHA = 3 /* GP relative 32 bit */
    R_ALPHA_LITERAL        R_ALPHA = 4 /* GP relative 16 bit w/opti
    R_ALPHA_LITUSE         R_ALPHA = 5 /* Optimization hint for LIT
    R_ALPHA_GPDISP         R_ALPHA = 6 /* Add displacement to GP */
    R_ALPHA_BRADDR         R_ALPHA = 7 /* PC+4 relative 23 bit shif
    R_ALPHA_HINT           R_ALPHA = 8 /* PC+4 relative 16 bit shif
    R_ALPHA_SREL16         R_ALPHA = 9 /* PC relative 16 bit */
    R_ALPHA_SREL32         R_ALPHA = 10 /* PC relative 32 bit */
    R_ALPHA_SREL64         R_ALPHA = 11 /* PC relative 64 bit */
    R_ALPHA_OP_PUSH        R_ALPHA = 12 /* OP stack push */
    R_ALPHA_OP_STORE       R_ALPHA = 13 /* OP stack pop and store */
    R_ALPHA_OP_PSUB        R_ALPHA = 14 /* OP stack subtract */
    R_ALPHA_OP_PRSHIFT     R_ALPHA = 15 /* OP stack right shift */
    R_ALPHA_GPVALUE        R_ALPHA = 16
    R_ALPHA_GPRELHIGH      R_ALPHA = 17
    R_ALPHA_GPRELLOW      R_ALPHA = 18
    R_ALPHA_IMMED_GP_16    R_ALPHA = 19
    R_ALPHA_IMMED_GP_HI32  R_ALPHA = 20
    R_ALPHA_IMMED_SCN_HI32 R_ALPHA = 21
    R_ALPHA_IMMED_BR_HI32  R_ALPHA = 22
    R_ALPHA_IMMED_LO32     R_ALPHA = 23
    R_ALPHA_COPY           R_ALPHA = 24 /* Copy symbol at runtime */
    R_ALPHA_GLOB_DAT       R_ALPHA = 25 /* Create GOT entry */
    R_ALPHA_JMP_SLOT       R_ALPHA = 26 /* Create PLT entry */
    R_ALPHA_RELATIVE       R_ALPHA = 27 /* Adjust by program base */
)
```

func (R_ALPHA) [GoString](#)

```
func (i R_ALPHA) GoString() string
```

func (R_ALPHA) [String](#)

```
func (i R_ALPHA) String() string
```

type [R_ARM](#)

```
type R_ARM int
```

Relocation types for ARM.

```
const (  
    R_ARM_NONE           R_ARM = 0 /* No relocation. */  
    R_ARM_PC24           R_ARM = 1  
    R_ARM_ABS32          R_ARM = 2  
    R_ARM_REL32          R_ARM = 3  
    R_ARM_PC13           R_ARM = 4  
    R_ARM_ABS16          R_ARM = 5  
    R_ARM_ABS12          R_ARM = 6  
    R_ARM_THM_ABS5       R_ARM = 7  
    R_ARM_ABS8           R_ARM = 8  
    R_ARM_SBREL32        R_ARM = 9  
    R_ARM_THM_PC22       R_ARM = 10  
    R_ARM_THM_PC8        R_ARM = 11  
    R_ARM_AMP_VCALL9     R_ARM = 12  
    R_ARM_SWI24           R_ARM = 13  
    R_ARM_THM_SWI8       R_ARM = 14  
    R_ARM_XPC25           R_ARM = 15  
    R_ARM_THM_XPC22      R_ARM = 16  
    R_ARM_COPY           R_ARM = 20 /* Copy data from shared object.  
    R_ARM_GLOB_DAT        R_ARM = 21 /* Set GOT entry to data address.  
    R_ARM_JUMP_SLOT      R_ARM = 22 /* Set GOT entry to code address.  
    R_ARM_RELATIVE        R_ARM = 23 /* Add load address of shared obj  
    R_ARM_GOTOFF          R_ARM = 24 /* Add GOT-relative symbol address  
    R_ARM_GOTPC           R_ARM = 25 /* Add PC-relative GOT table address  
    R_ARM_GOT32           R_ARM = 26 /* Add PC-relative GOT offset. */  
    R_ARM_PLT32           R_ARM = 27 /* Add PC-relative PLT offset. */  
    R_ARM_GNU_VTENTRY     R_ARM = 100  
    R_ARM_GNU_VTINHERIT  R_ARM = 101  
    R_ARM_RSBREL32        R_ARM = 250  
    R_ARM_THM_RPC22       R_ARM = 251  
    R_ARM_RREL32          R_ARM = 252  
    R_ARM_RABS32          R_ARM = 253  
    R_ARM_RPC24           R_ARM = 254  
    R_ARM_RBASE           R_ARM = 255  
)
```

func (R_ARM) [GoString](#)

```
func (i R_ARM) GoString() string
```

func (R_ARM) [String](#)

func (i R_ARM) String() string

type [R_PPC](#)

type R_PPC int

Relocation types for PowerPC.

```
const (
    R_PPC_NONE           R_PPC = 0 /* No relocation. */
    R_PPC_ADDR32         R_PPC = 1
    R_PPC_ADDR24         R_PPC = 2
    R_PPC_ADDR16         R_PPC = 3
    R_PPC_ADDR16_LO     R_PPC = 4
    R_PPC_ADDR16_HI     R_PPC = 5
    R_PPC_ADDR16_HA     R_PPC = 6
    R_PPC_ADDR14        R_PPC = 7
    R_PPC_ADDR14_BRTAKEN R_PPC = 8
    R_PPC_ADDR14_BRNTAKEN R_PPC = 9
    R_PPC_REL24         R_PPC = 10
    R_PPC_REL14         R_PPC = 11
    R_PPC_REL14_BRTAKEN R_PPC = 12
    R_PPC_REL14_BRNTAKEN R_PPC = 13
    R_PPC_GOT16         R_PPC = 14
    R_PPC_GOT16_LO     R_PPC = 15
    R_PPC_GOT16_HI     R_PPC = 16
    R_PPC_GOT16_HA     R_PPC = 17
    R_PPC_PLTREL24     R_PPC = 18
    R_PPC_COPY         R_PPC = 19
    R_PPC_GLOB_DAT     R_PPC = 20
    R_PPC_JMP_SLOT     R_PPC = 21
    R_PPC_RELATIVE     R_PPC = 22
    R_PPC_LOCAL24PC    R_PPC = 23
    R_PPC_UADDR32      R_PPC = 24
    R_PPC_UADDR16      R_PPC = 25
    R_PPC_REL32        R_PPC = 26
    R_PPC_PLT32        R_PPC = 27
    R_PPC_PLTREL32     R_PPC = 28
    R_PPC_PLT16_LO     R_PPC = 29
    R_PPC_PLT16_HI     R_PPC = 30
    R_PPC_PLT16_HA     R_PPC = 31
    R_PPC_SDAREL16     R_PPC = 32
    R_PPC_SECTOFF      R_PPC = 33
    R_PPC_SECTOFF_LO   R_PPC = 34
    R_PPC_SECTOFF_HI   R_PPC = 35
    R_PPC_SECTOFF_HA   R_PPC = 36
    R_PPC_TLS          R_PPC = 67
    R_PPC_DTPMOD32     R_PPC = 68
    R_PPC_TPREL16      R_PPC = 69
```

R_PPC_TPREL16_LO	R_PPC = 70
R_PPC_TPREL16_HI	R_PPC = 71
R_PPC_TPREL16_HA	R_PPC = 72
R_PPC_TPREL32	R_PPC = 73
R_PPC_DTPREL16	R_PPC = 74
R_PPC_DTPREL16_LO	R_PPC = 75
R_PPC_DTPREL16_HI	R_PPC = 76
R_PPC_DTPREL16_HA	R_PPC = 77
R_PPC_DTPREL32	R_PPC = 78
R_PPC_GOT_TLSGD16	R_PPC = 79
R_PPC_GOT_TLSGD16_LO	R_PPC = 80
R_PPC_GOT_TLSGD16_HI	R_PPC = 81
R_PPC_GOT_TLSGD16_HA	R_PPC = 82
R_PPC_GOT_TLSLD16	R_PPC = 83
R_PPC_GOT_TLSLD16_LO	R_PPC = 84
R_PPC_GOT_TLSLD16_HI	R_PPC = 85
R_PPC_GOT_TLSLD16_HA	R_PPC = 86
R_PPC_GOT_TPREL16	R_PPC = 87
R_PPC_GOT_TPREL16_LO	R_PPC = 88
R_PPC_GOT_TPREL16_HI	R_PPC = 89
R_PPC_GOT_TPREL16_HA	R_PPC = 90
R_PPC_EMB_NADDR32	R_PPC = 101
R_PPC_EMB_NADDR16	R_PPC = 102
R_PPC_EMB_NADDR16_LO	R_PPC = 103
R_PPC_EMB_NADDR16_HI	R_PPC = 104
R_PPC_EMB_NADDR16_HA	R_PPC = 105
R_PPC_EMB_SDAI16	R_PPC = 106
R_PPC_EMB_SDA2I16	R_PPC = 107
R_PPC_EMB_SDA2REL	R_PPC = 108
R_PPC_EMB_SDA21	R_PPC = 109
R_PPC_EMB_MRKREF	R_PPC = 110
R_PPC_EMB_RELSEC16	R_PPC = 111
R_PPC_EMB_RELST_LO	R_PPC = 112
R_PPC_EMB_RELST_HI	R_PPC = 113
R_PPC_EMB_RELST_HA	R_PPC = 114
R_PPC_EMB_BIT_FLD	R_PPC = 115
R_PPC_EMB_RELSDA	R_PPC = 116

)

func (R_PPC) [GoString](#)

func (i R_PPC) GoString() string

func (R_PPC) [String](#)

func (i R_PPC) String() string

type R_SPARC

type R_SPARC int

Relocation types for SPARC.

```
const (
    R_SPARC_NONE      R_SPARC = 0
    R_SPARC_8         R_SPARC = 1
    R_SPARC_16        R_SPARC = 2
    R_SPARC_32        R_SPARC = 3
    R_SPARC_DISP8     R_SPARC = 4
    R_SPARC_DISP16    R_SPARC = 5
    R_SPARC_DISP32    R_SPARC = 6
    R_SPARC_WDISP30   R_SPARC = 7
    R_SPARC_WDISP22   R_SPARC = 8
    R_SPARC_HI22      R_SPARC = 9
    R_SPARC_22        R_SPARC = 10
    R_SPARC_13        R_SPARC = 11
    R_SPARC_L010      R_SPARC = 12
    R_SPARC_GOT10     R_SPARC = 13
    R_SPARC_GOT13     R_SPARC = 14
    R_SPARC_GOT22     R_SPARC = 15
    R_SPARC_PC10      R_SPARC = 16
    R_SPARC_PC22      R_SPARC = 17
    R_SPARC_WPLT30    R_SPARC = 18
    R_SPARC_COPY      R_SPARC = 19
    R_SPARC_GLOB_DAT  R_SPARC = 20
    R_SPARC_JMP_SLOT  R_SPARC = 21
    R_SPARC_RELATIVE  R_SPARC = 22
    R_SPARC_UA32      R_SPARC = 23
    R_SPARC_PLT32     R_SPARC = 24
    R_SPARC_HIPLT22   R_SPARC = 25
    R_SPARC_LOPLT10   R_SPARC = 26
    R_SPARC_PCPLT32   R_SPARC = 27
    R_SPARC_PCPLT22   R_SPARC = 28
    R_SPARC_PCPLT10   R_SPARC = 29
    R_SPARC_10        R_SPARC = 30
    R_SPARC_11        R_SPARC = 31
    R_SPARC_64        R_SPARC = 32
    R_SPARC_OL010     R_SPARC = 33
    R_SPARC_HH22      R_SPARC = 34
    R_SPARC_HM10      R_SPARC = 35
    R_SPARC_LM22      R_SPARC = 36
    R_SPARC_PC_HH22   R_SPARC = 37
    R_SPARC_PC_HM10   R_SPARC = 38
    R_SPARC_PC_LM22   R_SPARC = 39
```

```
R_SPARC_WDISP16 R_SPARC = 40
R_SPARC_WDISP19 R_SPARC = 41
R_SPARC_GLOB_JMP R_SPARC = 42
R_SPARC_7       R_SPARC = 43
R_SPARC_5       R_SPARC = 44
R_SPARC_6       R_SPARC = 45
R_SPARC_DISP64 R_SPARC = 46
R_SPARC_PLT64  R_SPARC = 47
R_SPARC_HIX22  R_SPARC = 48
R_SPARC_LOX10  R_SPARC = 49
R_SPARC_H44    R_SPARC = 50
R_SPARC_M44    R_SPARC = 51
R_SPARC_L44    R_SPARC = 52
R_SPARC_REGISTER R_SPARC = 53
R_SPARC_UA64   R_SPARC = 54
R_SPARC_UA16   R_SPARC = 55
```

)

func (R_SPARC) [GoString](#)

```
func (i R_SPARC) GoString() string
```

func (R_SPARC) [String](#)

```
func (i R_SPARC) String() string
```

type [R_X86_64](#)

```
type R_X86_64 int
```

Relocation types for x86-64.

```
const (
    R_X86_64_NONE      R_X86_64 = 0 /* No relocation. */
    R_X86_64_64        R_X86_64 = 1 /* Add 64 bit symbol value. */
    R_X86_64_PC32      R_X86_64 = 2 /* PC-relative 32 bit signed sym
    R_X86_64_GOT32     R_X86_64 = 3 /* PC-relative 32 bit GOT offset
    R_X86_64_PLT32     R_X86_64 = 4 /* PC-relative 32 bit PLT offset
    R_X86_64_COPY      R_X86_64 = 5 /* Copy data from shared object.
    R_X86_64_GLOB_DAT  R_X86_64 = 6 /* Set GOT entry to data address
    R_X86_64_JMP_SLOT  R_X86_64 = 7 /* Set GOT entry to code address
    R_X86_64_RELATIVE  R_X86_64 = 8 /* Add load address of shared ob
    R_X86_64_GOTPCREL  R_X86_64 = 9 /* Add 32 bit signed pcrel offse
    R_X86_64_32        R_X86_64 = 10 /* Add 32 bit zero extended symb
    R_X86_64_32S       R_X86_64 = 11 /* Add 32 bit sign extended symb
    R_X86_64_16        R_X86_64 = 12 /* Add 16 bit zero extended symb
    R_X86_64_PC16      R_X86_64 = 13 /* Add 16 bit signed extended pc
    R_X86_64_8         R_X86_64 = 14 /* Add 8 bit zero extended symbo
    R_X86_64_PC8       R_X86_64 = 15 /* Add 8 bit signed extended pc
    R_X86_64_DTPMOD64  R_X86_64 = 16 /* ID of module containing symbo
    R_X86_64_DTPOFF64  R_X86_64 = 17 /* Offset in TLS block */
    R_X86_64_TPOFF64   R_X86_64 = 18 /* Offset in static TLS block */
    R_X86_64_TLSGD     R_X86_64 = 19 /* PC relative offset to GD GOT
    R_X86_64_TLSLD     R_X86_64 = 20 /* PC relative offset to LD GOT
    R_X86_64_DTPOFF32  R_X86_64 = 21 /* Offset in TLS block */
    R_X86_64_GOTTPOFF  R_X86_64 = 22 /* PC relative offset to IE GOT
    R_X86_64_TPOFF32   R_X86_64 = 23 /* Offset in static TLS block */
)
```

func (R_X86_64) [GoString](#)

```
func (i R_X86_64) GoString() string
```

func (R_X86_64) [String](#)

```
func (i R_X86_64) String() string
```

type [Rel32](#)

```
type Rel32 struct {  
    Off  uint32 /* Location to be relocated. */  
    Info uint32 /* Relocation type and symbol index. */  
}
```

ELF32 Relocations that don't need an addend field.

type [Rel64](#)

```
type Rel64 struct {  
    Off  uint64 /* Location to be relocated. */  
    Info uint64 /* Relocation type and symbol index. */  
}
```

ELF64 relocations that don't need an addend field.

type [Rela32](#)

```
type Rela32 struct {
    Off    uint32 /* Location to be relocated. */
    Info   uint32 /* Relocation type and symbol index. */
    Addend int32 /* Addend. */
}
```

ELF32 Relocations that need an addend field.

type [Rela64](#)

```
type Rela64 struct {
    Off    uint64 /* Location to be relocated. */
    Info   uint64 /* Relocation type and symbol index. */
    Addend int64 /* Addend. */
}
```

ELF64 relocations that need an addend field.

type [Section](#)

```
type Section struct {
    SectionHeader

    // Embed ReaderAt for ReadAt method.
    // Do not embed SectionReader directly
    // to avoid having Read and Seek.
    // If a client wants Read and Seek it must use
    // Open() to avoid fighting over the seek offset
    // with other clients.
    io.ReaderAt
    // contains filtered or unexported fields
}
```

A Section represents a single section in an ELF file.

func (***Section**) [Data](#)

```
func (s *Section) Data() ([]byte, error)
```

Data reads and returns the contents of the ELF section.

func (***Section**) [Open](#)

```
func (s *Section) Open() io.ReadSeeker
```

Open returns a new ReadSeeker reading the ELF section.

type [Section32](#)

```
type Section32 struct {
    Name      uint32 /* Section name (index into the section header) */
    Type      uint32 /* Section type. */
    Flags     uint32 /* Section flags. */
    Addr      uint32 /* Address in memory image. */
    Off       uint32 /* Offset in file. */
    Size      uint32 /* Size in bytes. */
    Link      uint32 /* Index of a related section. */
    Info      uint32 /* Depends on section type. */
    Addralign uint32 /* Alignment in bytes. */
    Entsize   uint32 /* Size of each entry in section. */
}
```

ELF32 Section header.

type [Section64](#)

```
type Section64 struct {
    Name      uint32 /* Section name (index into the section header
    Type      uint32 /* Section type. */
    Flags     uint64 /* Section flags. */
    Addr      uint64 /* Address in memory image. */
    Off       uint64 /* Offset in file. */
    Size      uint64 /* Size in bytes. */
    Link      uint32 /* Index of a related section. */
    Info      uint32 /* Depends on section type. */
    Addralign uint64 /* Alignment in bytes. */
    Entsize   uint64 /* Size of each entry in section. */
}
```

ELF64 Section header.

type [SectionFlag](#)

```
type SectionFlag uint32
```

Section flags.

```
const (
    SHF_WRITE           SectionFlag = 0x1        /* Section contain
    SHF_ALLOC           SectionFlag = 0x2        /* Section occupie
    SHF_EXECINSTR       SectionFlag = 0x4        /* Section contain
    SHF_MERGE           SectionFlag = 0x10       /* Section may be
    SHF_STRINGS         SectionFlag = 0x20       /* Section contain
    SHF_INFO_LINK       SectionFlag = 0x40       /* sh_info holds s
    SHF_LINK_ORDER      SectionFlag = 0x80       /* Special orderin
    SHF_OS_NONCONFORMING SectionFlag = 0x100     /* OS-specific pro
    SHF_GROUP           SectionFlag = 0x200     /* Member of secti
    SHF_TLS             SectionFlag = 0x400     /* Section contain
    SHF_MASKOS          SectionFlag = 0x0ff00000 /* OS-specific ser
    SHF_MASKPROC        SectionFlag = 0xf0000000 /* Processor-speci
)
```

func (SectionFlag) [GoString](#)

```
func (i SectionFlag) GoString() string
```

func (SectionFlag) [String](#)

```
func (i SectionFlag) String() string
```

type SectionHeader

```
type SectionHeader struct {  
    Name      string  
    Type      SectionType  
    Flags     SectionFlag  
    Addr      uint64  
    Offset    uint64  
    Size      uint64  
    Link      uint32  
    Info      uint32  
    Addralign uint64  
    Entsize   uint64  
}
```

A SectionHeader represents a single ELF section header.

type [SectionIndex](#)

```
type SectionIndex int
```

Special section indices.

```
const (  
    SHN_UNDEF      SectionIndex = 0      /* Undefined, missing, irrel  
    SHN_LORESERVE  SectionIndex = 0xff00 /* First of reserved range.  
    SHN_LOPROC     SectionIndex = 0xff00 /* First processor-specific.  
    SHN_HIPROC     SectionIndex = 0xff1f /* Last processor-specific.  
    SHN_LOOS       SectionIndex = 0xff20 /* First operating system-sp  
    SHN_HIOS       SectionIndex = 0xff3f /* Last operating system-spe  
    SHN_ABS        SectionIndex = 0xffff1 /* Absolute values. */  
    SHN_COMMON     SectionIndex = 0xffff2 /* Common data. */  
    SHN_XINDEX     SectionIndex = 0xffff /* Escape -- index stored el  
    SHN_HIRESERVE  SectionIndex = 0xffff /* Last of reserved range. *  
)
```

func (SectionIndex) [GoString](#)

```
func (i SectionIndex) GoString() string
```

func (SectionIndex) [String](#)

```
func (i SectionIndex) String() string
```

type [SectionType](#)

```
type SectionType uint32
```

Section type.

```
const (  
    SHT_NULL           SectionType = 0           /* inactive */  
    SHT_PROGBITS       SectionType = 1           /* program defined i  
    SHT_SYMTAB         SectionType = 2           /* symbol table sect  
    SHT_STRTAB         SectionType = 3           /* string table sect  
    SHT_RELA           SectionType = 4           /* relocation sectio  
    SHT_HASH           SectionType = 5           /* symbol hash table  
    SHT_DYNAMIC        SectionType = 6           /* dynamic section *  
    SHT_NOTE           SectionType = 7           /* note section */  
    SHT_NOBITS         SectionType = 8           /* no space section  
    SHT_REL            SectionType = 9           /* relocation sectio  
    SHT_SHLIB          SectionType = 10          /* reserved - purpos  
    SHT_DYNSYM         SectionType = 11          /* dynamic symbol ta  
    SHT_INIT_ARRAY     SectionType = 14          /* Initialization fu  
    SHT_FINI_ARRAY     SectionType = 15          /* Termination funct  
    SHT_PREINIT_ARRAY  SectionType = 16          /* Pre-initializatio  
    SHT_GROUP          SectionType = 17          /* Section group. */  
    SHT_SYMTAB_SHNDX   SectionType = 18          /* Section indexes (  
    SHT_LOOS           SectionType = 0x60000000 /* First of OS speci  
    SHT_GNU_ATTRIBUTES SectionType = 0x6fffffff /* GNU object attrib  
    SHT_GNU_HASH       SectionType = 0x6fffffff /* GNU hash table */  
    SHT_GNU_LIBLIST    SectionType = 0x6fffffff /* GNU prelink libra  
    SHT_GNU_VERDEF     SectionType = 0x6fffffff /* GNU version defin  
    SHT_GNU_VERNEED    SectionType = 0x6fffffff /* GNU version needs  
    SHT_GNU_VERSYM     SectionType = 0x6fffffff /* GNU version symbo  
    SHT_HIOS           SectionType = 0x6fffffff /* Last of OS specif  
    SHT_LOPROC         SectionType = 0x70000000 /* reserved range fo  
    SHT_HIPROC         SectionType = 0x7fffffff /* specific section  
    SHT_LOUSER         SectionType = 0x80000000 /* reserved range fo  
    SHT_HIUSER         SectionType = 0xffffffff /* specific indexes  
)
```

func (SectionType) [GoString](#)

```
func (i SectionType) GoString() string
```

func (SectionType) [String](#)

```
func (i SectionType) String() string
```

type [Sym32](#)

```
type Sym32 struct {  
    Name  uint32  
    Value uint32  
    Size  uint32  
    Info  uint8  
    Other uint8  
    Shndx uint16  
}
```

ELF32 Symbol.

type [Sym64](#)

```
type Sym64 struct {
    Name  uint32 /* String table index of name. */
    Info  uint8  /* Type and binding information. */
    Other uint8  /* Reserved (not used). */
    Shndx uint16 /* Section index of symbol. */
    Value uint64 /* Symbol value. */
    Size  uint64 /* Size of associated object. */
}
```

ELF64 symbol table entries.

type [SymBind](#)

```
type SymBind int
```

Symbol Binding - ELFNN_ST_BIND - st_info

```
const (  
    STB_LOCAL  SymBind = 0  /* Local symbol */  
    STB_GLOBAL SymBind = 1  /* Global symbol */  
    STB_WEAK   SymBind = 2  /* like global - lower precedence */  
    STB_LOOS   SymBind = 10 /* Reserved range for operating system *  
    STB_HIOS   SymBind = 12 /* specific semantics. */  
    STB_LOPROC SymBind = 13 /* reserved range for processor */  
    STB_HIPROC SymBind = 15 /* specific semantics. */  
)
```

func [ST_BIND](#)

```
func ST_BIND(info uint8) SymBind
```

func (SymBind) [GoString](#)

```
func (i SymBind) GoString() string
```

func (SymBind) [String](#)

```
func (i SymBind) String() string
```

type [SymType](#)

```
type SymType int
```

Symbol type - ELFNN_ST_TYPE - st_info

```
const (  
    STT_NOTYPE   SymType = 0 /* Unspecified type. */  
    STT_OBJECT   SymType = 1 /* Data object. */  
    STT_FUNC     SymType = 2 /* Function. */  
    STT_SECTION  SymType = 3 /* Section. */  
    STT_FILE     SymType = 4 /* Source file. */  
    STT_COMMON   SymType = 5 /* Uninitialized common block. */  
    STT_TLS      SymType = 6 /* TLS object. */  
    STT_LOOS     SymType = 10 /* Reserved range for operating system  
    STT_HIOS     SymType = 12 /* specific semantics. */  
    STT_LOPROC   SymType = 13 /* reserved range for processor */  
    STT_HIPROC   SymType = 15 /* specific semantics. */  
)
```

func [ST_TYPE](#)

```
func ST_TYPE(info uint8) SymType
```

func (SymType) [GoString](#)

```
func (i SymType) GoString() string
```

func (SymType) [String](#)

```
func (i SymType) String() string
```

type [SymVis](#)

```
type SymVis int
```

Symbol visibility - ELFNN_ST_VISIBILITY - st_other

```
const (  
    STV_DEFAULT    SymVis = 0x0 /* Default visibility (see binding).  
    STV_INTERNAL   SymVis = 0x1 /* Special meaning in relocatable obj  
    STV_HIDDEN     SymVis = 0x2 /* Not visible. */  
    STV_PROTECTED SymVis = 0x3 /* Visible but not preemptible. */  
)
```

func [ST_VISIBILITY](#)

```
func ST_VISIBILITY(other uint8) SymVis
```

func (SymVis) [GoString](#)

```
func (i SymVis) GoString() string
```

func (SymVis) [String](#)

```
func (i SymVis) String() string
```

type [Symbol](#)

```
type Symbol struct {  
    Name      string  
    Info, Other byte  
    Section   SectionIndex  
    Value, Size uint64  
}
```

A Symbol represents an entry in an ELF symbol table section.

type [Type](#)

```
type Type uint16
```

Type is found in Header.Type.

```
const (  
    ET_NONE    Type = 0      /* Unknown type. */  
    ET_REL     Type = 1      /* Relocatable. */  
    ET_EXEC    Type = 2      /* Executable. */  
    ET_DYN     Type = 3      /* Shared object. */  
    ET_CORE    Type = 4      /* Core file. */  
    ET_LOOS    Type = 0xfe00 /* First operating system specific. */  
    ET_HIOS    Type = 0xfeff /* Last operating system-specific. */  
    ET_LOPROC  Type = 0xff00 /* First processor-specific. */  
    ET_HIPROC  Type = 0xffff /* Last processor-specific. */  
)
```

func (Type) [GoString](#)

```
func (i Type) GoString() string
```

func (Type) [String](#)

```
func (i Type) String() string
```

type [Version](#)

```
type Version byte
```

Version is found in Header.Ident[EI_VERSION] and Header.Version.

```
const (  
    EV_NONE    Version = 0  
    EV_CURRENT Version = 1  
)
```

func (Version) [GoString](#)

```
func (i Version) GoString() string
```

func (Version) [String](#)

```
func (i Version) String() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package gosym

```
import "debug/gosym"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package gosym implements access to the Go symbol and line number tables embedded in Go binaries generated by the gc compilers.

Index

[type `DecodingError`](#)

[func \(e `*DecodingError`\) `Error\(\)` string](#)

[type `Func`](#)

[type `LineTable`](#)

[func `NewLineTable`\(data \[\]byte, text uint64\) `*LineTable`](#)

[func \(t `*LineTable`\) `LineToPC`\(line int, maxpc uint64\) uint64](#)

[func \(t `*LineTable`\) `PCToLine`\(pc uint64\) int](#)

[type `Obj`](#)

[type `Sym`](#)

[func \(s `*Sym`\) `BaseName\(\)` string](#)

[func \(s `*Sym`\) `PackageName\(\)` string](#)

[func \(s `*Sym`\) `ReceiverName\(\)` string](#)

[func \(s `*Sym`\) `Static\(\)` bool](#)

[type `Table`](#)

[func `NewTable`\(symtab \[\]byte, pcln `*LineTable`\) \(`*Table`, error\)](#)

[func \(t `*Table`\) `LineToPC`\(file string, line int\) \(pc uint64, fn `*Func`, error\)](#)

[func \(t `*Table`\) `LookupFunc`\(name string\) `*Func`](#)

[func \(t `*Table`\) `LookupSym`\(name string\) `*Sym`](#)

[func \(t `*Table`\) `PCToFunc`\(pc uint64\) `*Func`](#)

[func \(t `*Table`\) `PCToLine`\(pc uint64\) \(file string, line int, fn `*Func`\)](#)

[func \(t `*Table`\) `SymByAddr`\(addr uint64\) `*Sym`](#)

[type `UnknownFileError`](#)

[func \(e `UnknownFileError`\) `Error\(\)` string](#)

[type `UnknownLineError`](#)

[func \(e `*UnknownLineError`\) `Error\(\)` string](#)

Package files

[pclntab.go](#) [symtab.go](#)

type [DecodingError](#)

```
type DecodingError struct {  
    // contains filtered or unexported fields  
}
```

DecodingError represents an error during the decoding of the symbol table.

func ([*DecodingError](#)) [Error](#)

```
func (e *DecodingError) Error() string
```

type [Func](#)

```
type Func struct {
    Entry uint64
    *Sym
    End      uint64
    Params   []*Sym
    Locals   []*Sym
    FrameSize int
    LineTable *LineTable
    Obj      *Obj
}
```

A Func collects information about a single function.

type [LineTable](#)

```
type LineTable struct {  
    Data []byte  
    PC   uint64  
    Line int  
}
```

func [NewLineTable](#)

```
func NewLineTable(data []byte, text uint64) *LineTable
```

NewLineTable returns a new PC/line table corresponding to the encoded data. Text must be the start address of the corresponding text segment.

func (*LineTable) [LineToPC](#)

```
func (t *LineTable) LineToPC(line int, maxpc uint64) uint64
```

func (*LineTable) [PCToLine](#)

```
func (t *LineTable) PCToLine(pc uint64) int
```

type Obj

```
type Obj struct {  
    Funcs []Func  
    Paths []Sym  
}
```

An Obj represents a single object file.

type [Sym](#)

```
type Sym struct {
    Value  uint64
    Type   byte
    Name   string
    GoType uint64
    // If this symbol is a function symbol, the corresponding Func
    Func  *Func
}
```

A Sym represents a single symbol table entry.

func (*Sym) [BaseName](#)

```
func (s *Sym) BaseName() string
```

BaseName returns the symbol name without the package or receiver name.

func (*Sym) [PackageName](#)

```
func (s *Sym) PackageName() string
```

PackageName returns the package part of the symbol name, or the empty string if there is none.

func (*Sym) [ReceiverName](#)

```
func (s *Sym) ReceiverName() string
```

ReceiverName returns the receiver type name of this symbol, or the empty string if there is none.

func (*Sym) [Static](#)

```
func (s *Sym) Static() bool
```

Static returns whether this symbol is static (not visible outside its file).

type [Table](#)

```
type Table struct {
    Syms    []Sym
    Funcs   []Func
    Files   map[string]*Obj
    Objs    []Obj
}
```

Table represents a Go symbol table. It stores all of the symbols decoded from the program and provides methods to translate between symbols, names, and addresses.

func [NewTable](#)

```
func NewTable(symtab []byte, pcIn *LineTable) (*Table, error)
```

NewTable decodes the Go symbol table in data, returning an in-memory representation.

func (*Table) [LineToPC](#)

```
func (t *Table) LineToPC(file string, line int) (pc uint64, fn *Func)
```

LineToPC looks up the first program counter on the given line in the named file. Returns `UnknownPathError` or `UnknownLineError` if there is an error looking up this line.

func (*Table) [LookupFunc](#)

```
func (t *Table) LookupFunc(name string) *Func
```

LookupFunc returns the text, data, or bss symbol with the given name, or nil if no such symbol is found.

func (*Table) [LookupSym](#)

```
func (t *Table) LookupSym(name string) *Sym
```

LookupSym returns the text, data, or bss symbol with the given name, or nil if no such symbol is found.

func (*Table) [PCToFunc](#)

```
func (t *Table) PCToFunc(pc uint64) *Func
```

PCToFunc returns the function containing the program counter pc, or nil if there is no such function.

func (*Table) [PCToLine](#)

```
func (t *Table) PCToLine(pc uint64) (file string, line int, fn *Func)
```

PCToLine looks up line number information for a program counter. If there is no information, it returns fn == nil.

func (*Table) [SymByAddr](#)

```
func (t *Table) SymByAddr(addr uint64) *Sym
```

SymByAddr returns the text, data, or bss symbol starting at the given address.
TODO(rsc): Allow lookup by any address within the symbol.

type [UnknownFileError](#)

```
type UnknownFileError string
```

UnknownFileError represents a failure to find the specific file in the symbol table.

func (UnknownFileError) [Error](#)

```
func (e UnknownFileError) Error() string
```

type [UnknownLineError](#)

```
type UnknownLineError struct {  
    File string  
    Line int  
}
```

UnknownLineError represents a failure to map a line to a program counter, either because the line is beyond the bounds of the file or because there is no code on the given line.

func (*UnknownLineError) [Error](#)

```
func (e *UnknownLineError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package macho

```
import "debug/macho"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package macho implements access to Mach-O object files.

Index

Constants

type Cpu

func (i Cpu) GoString() string

func (i Cpu) String() string

type Dylib

type DylibCmd

type Dysymtab

type DysymtabCmd

type File

func NewFile(r io.ReaderAt) (*File, error)

func Open(name string) (*File, error)

func (f *File) Close() error

func (f *File) DWARF() (*dwarf.Data, error)

func (f *File) ImportedLibraries() ([]string, error)

func (f *File) ImportedSymbols() ([]string, error)

func (f *File) Section(name string) *Section

func (f *File) Segment(name string) *Segment

type FileHeader

type FormatError

func (e *FormatError) Error() string

type Load

type LoadBytes

func (b LoadBytes) Raw() []byte

type LoadCmd

func (i LoadCmd) GoString() string

func (i LoadCmd) String() string

type Nlist32

type Nlist64

type Regs386

type RegsAMD64

type Section

func (s *Section) Data() ([]byte, error)

func (s *Section) Open() io.ReadSeeker

type Section32

type Section64

[type SectionHeader](#)
[type Segment](#)
 [func \(s *Segment\) Data\(\) \(\[\]byte, error\)](#)
 [func \(s *Segment\) Open\(\) io.ReadSeeker](#)
[type Segment32](#)
[type Segment64](#)
[type SegmentHeader](#)
[type Symbol](#)
[type Symtab](#)
[type SymtabCmd](#)
[type Thread](#)
[type Type](#)

Package files

[file.go](#) [macho.go](#)

Constants

```
const (  
    Magic32 uint32 = 0xfeedface  
    Magic64 uint32 = 0xfeedfacf  
)
```

type [Cpu](#)

```
type Cpu uint32
```

A Cpu is a Mach-O cpu type.

```
const (  
    Cpu386    Cpu = 7  
    CpuAmd64 Cpu = Cpu386 + 1<<24  
)
```

func (Cpu) [GoString](#)

```
func (i Cpu) GoString() string
```

func (Cpu) [String](#)

```
func (i Cpu) String() string
```

type [Dylib](#)

```
type Dylib struct {  
    LoadBytes  
    Name          string  
    Time          uint32  
    CurrentVersion uint32  
    CompatVersion uint32  
}
```

A Dylib represents a Mach-O load dynamic library command.

type [DylibCmd](#)

```
type DylibCmd struct {  
    Cmd          LoadCmd  
    Len          uint32  
    Name        uint32  
    Time        uint32  
    CurrentVersion uint32  
    CompatVersion uint32  
}
```

A DylibCmd is a Mach-O load dynamic library command.

type [Dysymtab](#)

```
type Dysymtab struct {  
    LoadBytes  
    DysymtabCmd  
    IndirectSyms []uint32 // indices into Symtab.Syms  
}
```

A Dysymtab represents a Mach-O dynamic symbol table command.

type [DysymtabCmd](#)

```
type DysymtabCmd struct {
    Cmd          LoadCmd
    Len          uint32
    Ilocalsym    uint32
    Nlocalsym    uint32
    Iextdefsym   uint32
    Nextdefsym   uint32
    Iundefsym    uint32
    Nundefsym    uint32
    Tocoffset    uint32
    Ntoc         uint32
    Modtaboff    uint32
    Nmodtab      uint32
    Extrefsymoff uint32
    Nextrefsyms  uint32
    Indirectsymoff uint32
    Nindirectsyms uint32
    Extreloff    uint32
    Nextrel      uint32
    Locreloff    uint32
    Nlocrel      uint32
}
```

A DysymtabCmd is a Mach-O dynamic symbol table command.

type [File](#)

```
type File struct {
    FileHeader
    ByteOrder binary.ByteOrder
    Loads      []Load
    Sections   []*Section

    Symtab     *Symtab
    Dysymtab   *Dysymtab
    // contains filtered or unexported fields
}
```

A File represents an open Mach-O file.

func [NewFile](#)

```
func NewFile(r io.ReaderAt) (*File, error)
```

NewFile creates a new File for accessing a Mach-O binary in an underlying reader. The Mach-O binary is expected to start at position 0 in the ReaderAt.

func [Open](#)

```
func Open(name string) (*File, error)
```

Open opens the named file using os.Open and prepares it for use as a Mach-O binary.

func (*File) [Close](#)

```
func (f *File) Close() error
```

Close closes the File. If the File was created using NewFile directly instead of Open, Close has no effect.

func (*File) [DWARF](#)

```
func (f *File) DWARF() (*dwarf.Data, error)
```

DWARF returns the DWARF debug information for the Mach-O file.

func (*File) [ImportedLibraries](#)

```
func (f *File) ImportedLibraries() ([]string, error)
```

ImportedLibraries returns the paths of all libraries referred to by the binary f that are expected to be linked with the binary at dynamic link time.

func (*File) [ImportedSymbols](#)

```
func (f *File) ImportedSymbols() ([]string, error)
```

ImportedSymbols returns the names of all symbols referred to by the binary f that are expected to be satisfied by other libraries at dynamic load time.

func (*File) [Section](#)

```
func (f *File) Section(name string) *Section
```

Section returns the first section with the given name, or nil if no such section exists.

func (*File) [Segment](#)

```
func (f *File) Segment(name string) *Segment
```

Segment returns the first Segment with the given name, or nil if no such segment exists.

type [FileHeader](#)

```
type FileHeader struct {  
    Magic    uint32  
    Cpu      Cpu  
    SubCpu   uint32  
    Type     Type  
    Ncmd     uint32  
    Cmdsz    uint32  
    Flags    uint32  
}
```

A FileHeader represents a Mach-O file header.

type [FormatError](#)

```
type FormatError struct {  
    // contains filtered or unexported fields  
}
```

func ([*FormatError](#)) [Error](#)

```
func (e *FormatError) Error() string
```

type [Load](#)

```
type Load interface {  
    Raw() []byte  
}
```

A Load represents any Mach-O load command.

type [LoadBytes](#)

```
type LoadBytes []byte
```

A LoadBytes is the uninterpreted bytes of a Mach-O load command.

func (LoadBytes) Raw

```
func (b LoadBytes) Raw() []byte
```

type [LoadCmd](#)

```
type LoadCmd uint32
```

A LoadCmd is a Mach-O load command.

```
const (  
    LoadCmdSegment      LoadCmd = 1  
    LoadCmdSymtab       LoadCmd = 2  
    LoadCmdThread       LoadCmd = 4  
    LoadCmdUnixThread   LoadCmd = 5 // thread+stack  
    LoadCmdDysymtab     LoadCmd = 11  
    LoadCmdDylib        LoadCmd = 12  
    LoadCmdDylinker     LoadCmd = 15  
    LoadCmdSegment64    LoadCmd = 25  
)
```

func (LoadCmd) [GoString](#)

```
func (i LoadCmd) GoString() string
```

func (LoadCmd) [String](#)

```
func (i LoadCmd) String() string
```

type [Nlist32](#)

```
type Nlist32 struct {  
    Name  uint32  
    Type  uint8  
    Sect  uint8  
    Desc  uint16  
    Value uint32  
}
```

An Nlist32 is a Mach-O 32-bit symbol table entry.

type [Nlist64](#)

```
type Nlist64 struct {  
    Name  uint32  
    Type  uint8  
    Sect  uint8  
    Desc  uint16  
    Value uint64  
}
```

An Nlist64 is a Mach-O 64-bit symbol table entry.

type [Regs386](#)

```
type Regs386 struct {
    AX    uint32
    BX    uint32
    CX    uint32
    DX    uint32
    DI    uint32
    SI    uint32
    BP    uint32
    SP    uint32
    SS    uint32
    FLAGS uint32
    IP    uint32
    CS    uint32
    DS    uint32
    ES    uint32
    FS    uint32
    GS    uint32
}
```

Regs386 is the Mach-O 386 register structure.

type [RegsAMD64](#)

```
type RegsAMD64 struct {  
    AX    uint64  
    BX    uint64  
    CX    uint64  
    DX    uint64  
    DI    uint64  
    SI    uint64  
    BP    uint64  
    SP    uint64  
    R8    uint64  
    R9    uint64  
    R10   uint64  
    R11   uint64  
    R12   uint64  
    R13   uint64  
    R14   uint64  
    R15   uint64  
    IP    uint64  
    FLAGS uint64  
    CS    uint64  
    FS    uint64  
    GS    uint64  
}
```

RegsAMD64 is the Mach-O AMD64 register structure.

type [Section](#)

```
type Section struct {
    SectionHeader

    // Embed ReaderAt for ReadAt method.
    // Do not embed SectionReader directly
    // to avoid having Read and Seek.
    // If a client wants Read and Seek it must use
    // Open() to avoid fighting over the seek offset
    // with other clients.
    io.ReaderAt
    // contains filtered or unexported fields
}
```

func (*Section) [Data](#)

```
func (s *Section) Data() ([]byte, error)
```

Data reads and returns the contents of the Mach-O section.

func (*Section) [Open](#)

```
func (s *Section) Open() io.ReadSeeker
```

Open returns a new ReadSeeker reading the Mach-O section.

type [Section32](#)

```
type Section32 struct {
    Name      [16]byte
    Seg       [16]byte
    Addr      uint32
    Size      uint32
    Offset    uint32
    Align     uint32
    Reloff    uint32
    Nreloc    uint32
    Flags     uint32
    Reserve1  uint32
    Reserve2  uint32
}
```

A Section32 is a 32-bit Mach-O section header.

type [Section64](#)

```
type Section64 struct {
    Name      [16]byte
    Seg       [16]byte
    Addr      uint64
    Size      uint64
    Offset    uint32
    Align     uint32
    Reloff    uint32
    Nreloc    uint32
    Flags     uint32
    Reserve1  uint32
    Reserve2  uint32
    Reserve3  uint32
}
```

A Section32 is a 64-bit Mach-O section header.

type SectionHeader

```
type SectionHeader struct {  
    Name    string  
    Seg     string  
    Addr    uint64  
    Size    uint64  
    Offset  uint32  
    Align   uint32  
    Reloff  uint32  
    Nreloc  uint32  
    Flags   uint32  
}
```

type [Segment](#)

```
type Segment struct {
    LoadBytes
    SegmentHeader

    // Embed ReaderAt for ReadAt method.
    // Do not embed SectionReader directly
    // to avoid having Read and Seek.
    // If a client wants Read and Seek it must use
    // Open() to avoid fighting over the seek offset
    // with other clients.
    io.ReaderAt
    // contains filtered or unexported fields
}
```

A Segment represents a Mach-O 32-bit or 64-bit load segment command.

func (*Segment) [Data](#)

```
func (s *Segment) Data() ([]byte, error)
```

Data reads and returns the contents of the segment.

func (*Segment) [Open](#)

```
func (s *Segment) Open() io.ReadSeeker
```

Open returns a new ReadSeeker reading the segment.

type Segment32

```
type Segment32 struct {  
    Cmd      LoadCmd  
    Len      uint32  
    Name     [16]byte  
    Addr     uint32  
    Memsz    uint32  
    Offset   uint32  
    Filesz   uint32  
    Maxprot  uint32  
    Prot     uint32  
    Nsect    uint32  
    Flag     uint32  
}
```

A Segment32 is a 32-bit Mach-O segment load command.

type Segment64

```
type Segment64 struct {  
    Cmd      LoadCmd  
    Len      uint32  
    Name     [16]byte  
    Addr     uint64  
    Memsz    uint64  
    Offset   uint64  
    Filesz   uint64  
    Maxprot  uint32  
    Prot     uint32  
    Nsect    uint32  
    Flag     uint32  
}
```

A Segment64 is a 64-bit Mach-O segment load command.

type SegmentHeader

```
type SegmentHeader struct {  
    Cmd      LoadCmd  
    Len      uint32  
    Name     string  
    Addr     uint64  
    Memsz    uint64  
    Offset   uint64  
    Filesz   uint64  
    Maxprot  uint32  
    Prot     uint32  
    Nsect    uint32  
    Flag     uint32  
}
```

A SegmentHeader is the header for a Mach-O 32-bit or 64-bit load segment command.

type [Symbol](#)

```
type Symbol struct {  
    Name  string  
    Type  uint8  
    Sect  uint8  
    Desc  uint16  
    Value uint64  
}
```

A Symbol is a Mach-O 32-bit or 64-bit symbol table entry.

type [Symtab](#)

```
type Symtab struct {  
    LoadBytes  
    SymtabCmd  
    Syms []Symbol  
}
```

A Symtab represents a Mach-O symbol table command.

type [SymtabCmd](#)

```
type SymtabCmd struct {  
    Cmd      LoadCmd  
    Len      uint32  
    Symoff   uint32  
    Nsyms    uint32  
    Stroff   uint32  
    Strsize  uint32  
}
```

A SymtabCmd is a Mach-O symbol table command.

type [Thread](#)

```
type Thread struct {  
    Cmd  LoadCmd  
    Len  uint32  
    Type uint32  
    Data []uint32  
}
```

A Thread is a Mach-O thread state command.

type [Type](#)

```
type Type uint32
```

A Type is a Mach-O file type, either an object or an executable.

```
const (  
    TypeObj   Type = 1  
    TypeExec  Type = 2  
)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package pe

```
import "debug/pe"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package pe implements access to PE (Microsoft Windows Portable Executable) files.

Index

Constants

type File

[func NewFile\(r io.ReaderAt\) \(*File, error\)](#)

[func Open\(name string\) \(*File, error\)](#)

[func \(f *File\) Close\(\) error](#)

[func \(f *File\) DWARF\(\) \(*dwarf.Data, error\)](#)

[func \(f *File\) ImportedLibraries\(\) \(\[\]string, error\)](#)

[func \(f *File\) ImportedSymbols\(\) \(\[\]string, error\)](#)

[func \(f *File\) Section\(name string\) *Section](#)

type FileHeader

type FormatError

[func \(e *FormatError\) Error\(\) string](#)

type ImportDirectory

type Section

[func \(s *Section\) Data\(\) \(\[\]byte, error\)](#)

[func \(s *Section\) Open\(\) io.ReadSeeker](#)

type SectionHeader

type SectionHeader32

Package files

[file.go](#) [pe.go](#)

Constants

```
const (  
    IMAGE_FILE_MACHINE_UNKNOWN = 0x0  
    IMAGE_FILE_MACHINE_AM33    = 0x1d3  
    IMAGE_FILE_MACHINE_AMD64   = 0x8664  
    IMAGE_FILE_MACHINE_ARM     = 0x1c0  
    IMAGE_FILE_MACHINE_EBC     = 0xebc  
    IMAGE_FILE_MACHINE_I386    = 0x14c  
    IMAGE_FILE_MACHINE_IA64    = 0x200  
    IMAGE_FILE_MACHINE_M32R    = 0x9041  
    IMAGE_FILE_MACHINE_MIPS16  = 0x266  
    IMAGE_FILE_MACHINE_MIPSFPU = 0x366  
    IMAGE_FILE_MACHINE_MIPSFPU16 = 0x466  
    IMAGE_FILE_MACHINE_POWERPC = 0x1f0  
    IMAGE_FILE_MACHINE_POWERPCFP = 0x1f1  
    IMAGE_FILE_MACHINE_R4000   = 0x166  
    IMAGE_FILE_MACHINE_SH3     = 0x1a2  
    IMAGE_FILE_MACHINE_SH3DSP  = 0x1a3  
    IMAGE_FILE_MACHINE_SH4     = 0x1a6  
    IMAGE_FILE_MACHINE_SH5     = 0x1a8  
    IMAGE_FILE_MACHINE_THUMB   = 0x1c2  
    IMAGE_FILE_MACHINE_WCEMIPSV2 = 0x169  
)
```

type [File](#)

```
type File struct {
    FileHeader
    Sections []*Section
    // contains filtered or unexported fields
}
```

A File represents an open PE file.

func [NewFile](#)

```
func NewFile(r io.ReaderAt) (*File, error)
```

NewFile creates a new File for accessing a PE binary in an underlying reader.

func [Open](#)

```
func Open(name string) (*File, error)
```

Open opens the named file using os.Open and prepares it for use as a PE binary.

func (***File**) [Close](#)

```
func (f *File) Close() error
```

Close closes the File. If the File was created using NewFile directly instead of Open, Close has no effect.

func (***File**) [DWARF](#)

```
func (f *File) DWARF() (*dwarf.Data, error)
```

func (***File**) [ImportedLibraries](#)

```
func (f *File) ImportedLibraries() ([]string, error)
```

ImportedLibraries returns the names of all libraries referred to by the binary f that are expected to be linked with the binary at dynamic link time.

func (*File) [ImportedSymbols](#)

```
func (f *File) ImportedSymbols() ([]string, error)
```

ImportedSymbols returns the names of all symbols referred to by the binary f that are expected to be satisfied by other libraries at dynamic load time. It does not return weak symbols.

func (*File) [Section](#)

```
func (f *File) Section(name string) *Section
```

Section returns the first section with the given name, or nil if no such section exists.

type [FileHeader](#)

```
type FileHeader struct {  
    Machine          uint16  
    NumberOfSections uint16  
    TimeDateStamp    uint32  
    PointerToSymbolTable uint32  
    NumberOfSymbols  uint32  
    SizeOfOptionalHeader uint16  
    Characteristics  uint16  
}
```

type [FormatError](#)

```
type FormatError struct {  
    // contains filtered or unexported fields  
}
```

func ([*FormatError](#)) [Error](#)

```
func (e *FormatError) Error() string
```

type ImportDirectory

```
type ImportDirectory struct {
    OriginalFirstThunk uint32
    TimeDateStamp      uint32
    ForwarderChain     uint32
    Name               uint32
    FirstThunk         uint32
    // contains filtered or unexported fields
}
```

type [Section](#)

```
type Section struct {
    SectionHeader

    // Embed ReaderAt for ReadAt method.
    // Do not embed SectionReader directly
    // to avoid having Read and Seek.
    // If a client wants Read and Seek it must use
    // Open() to avoid fighting over the seek offset
    // with other clients.
    io.ReaderAt
    // contains filtered or unexported fields
}
```

func (*Section) [Data](#)

```
func (s *Section) Data() ([]byte, error)
```

Data reads and returns the contents of the PE section.

func (*Section) [Open](#)

```
func (s *Section) Open() io.ReadSeeker
```

Open returns a new ReadSeeker reading the PE section.

type SectionHeader

```
type SectionHeader struct {  
    Name                string  
    VirtualSize         uint32  
    VirtualAddress      uint32  
    Size                uint32  
    Offset              uint32  
    PointerToRelocations uint32  
    PointerToLineNumbers uint32  
    NumberOfRelocations uint16  
    NumberOfLineNumbers  uint16  
    Characteristics     uint32  
}
```

type [SectionHeader32](#)

```
type SectionHeader32 struct {  
    Name           [8]uint8  
    VirtualSize    uint32  
    VirtualAddress uint32  
    SizeOfRawData  uint32  
    PointerToRawData uint32  
    PointerToRelocations uint32  
    PointerToLineNumbers uint32  
    NumberOfRelocations uint16  
    NumberOfLineNumbers uint16  
    Characteristics   uint32  
}
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg/encoding

Name	Synopsis
ascii85	Package ascii85 implements the ascii85 data encoding as used in the btoa tool and Adobe's PostScript and PDF document formats.
asn1	Package asn1 implements parsing of DER-encoded ASN.1 data structures, as defined in ITU-T Rec X.690.
base32	Package base32 implements base32 encoding as specified by RFC 4648.
base64	Package base64 implements base64 encoding as specified by RFC 4648.
binary	Package binary implements translation between numbers and byte sequences and encoding and decoding of varints.
csv	Package csv reads and writes comma-separated values (CSV) files.
gob	Package gob manages streams of gobs - binary values exchanged between an Encoder (transmitter) and a Decoder (receiver).
hex	Package hex implements hexadecimal encoding and decoding.
json	Package json implements encoding and decoding of JSON objects as defined in RFC 4627.
pem	Package pem implements the PEM data encoding, which originated in Privacy Enhanced Mail.
xml	Package xml implements a simple XML 1.0 parser that understands XML name spaces.

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package ascii85

```
import "encoding/ascii85"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `ascii85` implements the `ascii85` data encoding as used in the `btoa` tool and Adobe's PostScript and PDF document formats.

Index

[func Decode\(dst, src \[\]byte, flush bool\) \(ndst, nsrc int, err error\)](#)

[func Encode\(dst, src \[\]byte\) int](#)

[func MaxEncodedLen\(n int\) int](#)

[func NewDecoder\(r io.Reader\) io.Reader](#)

[func NewEncoder\(w io.Writer\) io.WriteCloser](#)

[type CorruptInputError](#)

[func \(e CorruptInputError\) Error\(\) string](#)

Package files

[ascii85.go](#)

func [Decode](#)

```
func Decode(dst, src []byte, flush bool) (ndst, nsrc int, err error)
```

Decode decodes src into dst, returning both the number of bytes written to dst and the number consumed from src. If src contains invalid ascii85 data, Decode will return the number of bytes successfully written and a `CorruptInputError`. Decode ignores space and control characters in src. Often, ascii85-encoded data is wrapped in `<~` and `~>` symbols. Decode expects these to have been stripped by the caller.

If flush is true, Decode assumes that src represents the end of the input stream and processes it completely rather than wait for the completion of another 32-bit block.

`NewDecoder` wraps an `io.Reader` interface around `Decode`.

func Encode

```
func Encode(dst, src []byte) int
```

Encode encodes src into at most MaxEncodedLen(len(src)) bytes of dst, returning the actual number of bytes written.

The encoding handles 4-byte chunks, using a special encoding for the last fragment, so Encode is not appropriate for use on individual blocks of a large data stream. Use NewEncoder() instead.

Often, ascii85-encoded data is wrapped in <~ and ~> symbols. Encode does not add these.

func MaxEncodedLen

```
func MaxEncodedLen(n int) int
```

MaxEncodedLen returns the maximum length of an encoding of n source bytes.

func [NewDecoder](#)

```
func NewDecoder(r io.Reader) io.Reader
```

NewDecoder constructs a new ascii85 stream decoder.

func [NewEncoder](#)

```
func NewEncoder(w io.Writer) io.WriteCloser
```

NewEncoder returns a new ascii85 stream encoder. Data written to the returned writer will be encoded and then written to w. Ascii85 encodings operate in 32-bit blocks; when finished writing, the caller must Close the returned encoder to flush any trailing partial block.

type [CorruptInputError](#)

```
type CorruptInputError int64
```

func (CorruptInputError) [Error](#)

```
func (e CorruptInputError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package asn1

```
import "encoding/asn1"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `asn1` implements parsing of DER-encoded ASN.1 data structures, as defined in ITU-T Rec X.690.

See also “A Layman's Guide to a Subset of ASN.1, BER, and DER,”
<http://luca.ntop.org/Teaching/Appunti/asn1.html>.

Index

[func Marshal\(val interface{}\) \(\[\]byte, error\)](#)
[func Unmarshal\(b \[\]byte, val interface{}\) \(rest \[\]byte, err error\)](#)
[func UnmarshalWithParams\(b \[\]byte, val interface{}, params string\) \(rest \[\]byte, err error\)](#)
[type BitString](#)
 [func \(b BitString\) At\(i int\) int](#)
 [func \(b BitString\) RightAlign\(\) \[\]byte](#)
[type Enumerated](#)
[type Flag](#)
[type ObjectIdentifier](#)
 [func \(oi ObjectIdentifier\) Equal\(other ObjectIdentifier\) bool](#)
[type RawContent](#)
[type RawValue](#)
[type StructuralError](#)
 [func \(e StructuralError\) Error\(\) string](#)
[type SyntaxError](#)
 [func \(e SyntaxError\) Error\(\) string](#)

Package files

[asn1.go](#) [common.go](#) [marshal.go](#)

func [Marshal](#)

```
func Marshal(val interface{}) ([]byte, error)
```

Marshal returns the ASN.1 encoding of val.

func [Unmarshal](#)

```
func Unmarshal(b []byte, val interface{}) (rest []byte, err error)
```

Unmarshal parses the DER-encoded ASN.1 data structure `b` and uses the `reflect` package to fill in an arbitrary value pointed at by `val`. Because Unmarshal uses the `reflect` package, the structs being written to must use upper case field names.

An ASN.1 INTEGER can be written to an `int`, `int32`, `int64`, or `*big.Int` (from the `math/big` package). If the encoded value does not fit in the Go type, Unmarshal returns a parse error.

An ASN.1 BIT STRING can be written to a `BitString`.

An ASN.1 OCTET STRING can be written to a `[]byte`.

An ASN.1 OBJECT IDENTIFIER can be written to an `ObjectIdentifier`.

An ASN.1 ENUMERATED can be written to an `Enumerated`.

An ASN.1 UTCTIME or GENERALIZEDTIME can be written to a `time.Time`.

An ASN.1 PrintableString or IA5String can be written to a `string`.

Any of the above ASN.1 values can be written to an `interface{}`. The value stored in the interface has the corresponding Go type. For integers, that type is `int64`.

An ASN.1 SEQUENCE OF `x` or SET OF `x` can be written to a slice if an `x` can be written to the slice's element type.

An ASN.1 SEQUENCE or SET can be written to a struct if each of the elements in the sequence can be written to the corresponding element in the struct.

The following tags on struct fields have special meaning to Unmarshal:

<code>optional</code>	marks the field as ASN.1 OPTIONAL
<code>[explicit] tag:x</code>	specifies the ASN.1 tag number; implies ASN.
<code>default:x</code>	sets the default value for optional integer

If the type of the first field of a structure is RawContent then the raw ASN1 contents of the struct will be stored in it.

Other ASN.1 types are not supported; if it encounters them, Unmarshal returns a parse error.

func UnmarshalWithParams

```
func UnmarshalWithParams(b []byte, val interface{}, params string) (
```

UnmarshalWithParams allows field parameters to be specified for the top-level element. The form of the params is the same as the field tags.

type [BitString](#)

```
type BitString struct {  
    Bytes      []byte // bits packed into bytes.  
    BitLength int    // length in bits.  
}
```

BitString is the structure to use when you want an ASN.1 BIT STRING type. A bit string is padded up to the nearest byte in memory and the number of valid bits is recorded. Padding bits will be zero.

func (BitString) [At](#)

```
func (b BitString) At(i int) int
```

At returns the bit at the given index. If the index is out of range it returns false.

func (BitString) [RightAlign](#)

```
func (b BitString) RightAlign() []byte
```

RightAlign returns a slice where the padding bits are at the beginning. The slice may share memory with the BitString.

type Enumerated

type Enumerated int

An Enumerated is represented as a plain int.

type **Flag**

```
type Flag bool
```

A Flag accepts any data and is set to true if present.

type [ObjectIdentifier](#)

```
type ObjectIdentifier []int
```

An ObjectIdentifier represents an ASN.1 OBJECT IDENTIFIER.

func (ObjectIdentifier) [Equal](#)

```
func (oi ObjectIdentifier) Equal(other ObjectIdentifier) bool
```

Equal returns true iff oi and other represent the same identifier.

type [RawContent](#)

```
type RawContent []byte
```

RawContent is used to signal that the undecoded, DER data needs to be preserved for a struct. To use it, the first field of the struct must have this type. It's an error for any of the other fields to have this type.

type [RawValue](#)

```
type RawValue struct {  
    Class, Tag int  
    IsCompound bool  
    Bytes      []byte  
    FullBytes  []byte // includes the tag and length  
}
```

A RawValue represents an undecoded ASN.1 object.

type [StructuralError](#)

```
type StructuralError struct {  
    Msg string  
}
```

A StructuralError suggests that the ASN.1 data is valid, but the Go type which is receiving it doesn't match.

func (StructuralError) [Error](#)

```
func (e StructuralError) Error() string
```

type [SyntaxError](#)

```
type SyntaxError struct {  
    Msg string  
}
```

A SyntaxError suggests that the ASN.1 data is invalid.

func (SyntaxError) [Error](#)

```
func (e SyntaxError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package base32

```
import "encoding/base32"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package base32 implements base32 encoding as specified by RFC 4648.

Index

Variables

[func NewDecoder\(enc *Encoding, r io.Reader\) io.Reader](#)

[func NewEncoder\(enc *Encoding, w io.Writer\) io.WriteCloser](#)

[type CorruptInputError](#)

[func \(e CorruptInputError\) Error\(\) string](#)

[type Encoding](#)

[func NewEncoding\(encoder string\) *Encoding](#)

[func \(enc *Encoding\) Decode\(dst, src \[\]byte\) \(n int, err error\)](#)

[func \(enc *Encoding\) DecodeString\(s string\) \(\[\]byte, error\)](#)

[func \(enc *Encoding\) DecodedLen\(n int\) int](#)

[func \(enc *Encoding\) Encode\(dst, src \[\]byte\)](#)

[func \(enc *Encoding\) EncodeToString\(src \[\]byte\) string](#)

[func \(enc *Encoding\) EncodedLen\(n int\) int](#)

Package files

[base32.go](#)

Variables

```
var HexEncoding = NewEncoding(encodeHex)
```

HexEncoding is the “Extended Hex Alphabet” defined in RFC 4648. It is typically used in DNS.

```
var StdEncoding = NewEncoding(encodeStd)
```

StdEncoding is the standard base32 encoding, as defined in RFC 4648.

func [NewDecoder](#)

```
func NewDecoder(enc *Encoding, r io.Reader) io.Reader
```

NewDecoder constructs a new base32 stream decoder.

func [NewEncoder](#)

```
func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser
```

NewEncoder returns a new base32 stream encoder. Data written to the returned writer will be encoded using enc and then written to w. Base32 encodings operate in 5-byte blocks; when finished writing, the caller must Close the returned encoder to flush any partially written blocks.

type CorruptInputError

type CorruptInputError int64

func (CorruptInputError) Error

func (e CorruptInputError) Error() string

type [Encoding](#)

```
type Encoding struct {  
    // contains filtered or unexported fields  
}
```

An Encoding is a radix 32 encoding/decoding scheme, defined by a 32-character alphabet. The most common is the "base32" encoding introduced for SASL GSSAPI and standardized in RFC 4648. The alternate "base32hex" encoding is used in DNSSEC.

func [NewEncoding](#)

```
func NewEncoding(encoder string) *Encoding
```

NewEncoding returns a new Encoding defined by the given alphabet, which must be a 32-byte string.

func (*Encoding) [Decode](#)

```
func (enc *Encoding) Decode(dst, src []byte) (n int, err error)
```

Decode decodes src using the encoding enc. It writes at most DecodedLen(len(src)) bytes to dst and returns the number of bytes written. If src contains invalid base32 data, it will return the number of bytes successfully written and CorruptInputError. New line characters (\r and \n) are ignored.

func (*Encoding) [DecodeString](#)

```
func (enc *Encoding) DecodeString(s string) ([]byte, error)
```

DecodeString returns the bytes represented by the base32 string s.

func (*Encoding) [DecodedLen](#)

```
func (enc *Encoding) DecodedLen(n int) int
```

DecodedLen returns the maximum length in bytes of the decoded data corresponding to n bytes of base32-encoded data.

func (*Encoding) [Encode](#)

```
func (enc *Encoding) Encode(dst, src []byte)
```

Encode encodes src using the encoding enc, writing EncodedLen(len(src)) bytes to dst.

The encoding pads the output to a multiple of 8 bytes, so Encode is not appropriate for use on individual blocks of a large data stream. Use NewEncoder() instead.

func (*Encoding) [EncodeToString](#)

```
func (enc *Encoding) EncodeToString(src []byte) string
```

EncodeToString returns the base32 encoding of src.

func (*Encoding) [EncodedLen](#)

```
func (enc *Encoding) EncodedLen(n int) int
```

EncodedLen returns the length in bytes of the base32 encoding of an input buffer of length n.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package base64

```
import "encoding/base64"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package base64 implements base64 encoding as specified by RFC 4648.

Index

Variables

[func NewDecoder\(enc *Encoding, r io.Reader\) io.Reader](#)

[func NewEncoder\(enc *Encoding, w io.Writer\) io.WriteCloser](#)

[type CorruptInputError](#)

[func \(e CorruptInputError\) Error\(\) string](#)

[type Encoding](#)

[func NewEncoding\(encoder string\) *Encoding](#)

[func \(enc *Encoding\) Decode\(dst, src \[\]byte\) \(n int, err error\)](#)

[func \(enc *Encoding\) DecodeString\(s string\) \(\[\]byte, error\)](#)

[func \(enc *Encoding\) DecodedLen\(n int\) int](#)

[func \(enc *Encoding\) Encode\(dst, src \[\]byte\)](#)

[func \(enc *Encoding\) EncodeToString\(src \[\]byte\) string](#)

[func \(enc *Encoding\) EncodedLen\(n int\) int](#)

Package files

[base64.go](#)

Variables

```
var StdEncoding = NewEncoding(encodeStd)
```

StdEncoding is the standard base64 encoding, as defined in RFC 4648.

```
var URLEncoding = NewEncoding(encodeURL)
```

URLEncoding is the alternate base64 encoding defined in RFC 4648. It is typically used in URLs and file names.

func [NewDecoder](#)

```
func NewDecoder(enc *Encoding, r io.Reader) io.Reader
```

NewDecoder constructs a new base64 stream decoder.

func [NewEncoder](#)

```
func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser
```

NewEncoder returns a new base64 stream encoder. Data written to the returned writer will be encoded using enc and then written to w. Base64 encodings operate in 4-byte blocks; when finished writing, the caller must Close the returned encoder to flush any partially written blocks.

type CorruptInputError

type CorruptInputError int64

func (CorruptInputError) Error

func (e CorruptInputError) Error() string

type [Encoding](#)

```
type Encoding struct {  
    // contains filtered or unexported fields  
}
```

An Encoding is a radix 64 encoding/decoding scheme, defined by a 64-character alphabet. The most common encoding is the "base64" encoding defined in RFC 4648 and used in MIME (RFC 2045) and PEM (RFC 1421). RFC 4648 also defines an alternate encoding, which is the standard encoding with - and _ substituted for + and /.

func [NewEncoding](#)

```
func NewEncoding(encoder string) *Encoding
```

NewEncoding returns a new Encoding defined by the given alphabet, which must be a 64-byte string.

func (*Encoding) [Decode](#)

```
func (enc *Encoding) Decode(dst, src []byte) (n int, err error)
```

Decode decodes src using the encoding enc. It writes at most DecodedLen(len(src)) bytes to dst and returns the number of bytes written. If src contains invalid base64 data, it will return the number of bytes successfully written and CorruptInputError. New line characters (\r and \n) are ignored.

func (*Encoding) [DecodeString](#)

```
func (enc *Encoding) DecodeString(s string) ([]byte, error)
```

DecodeString returns the bytes represented by the base64 string s.

func (*Encoding) [DecodedLen](#)

```
func (enc *Encoding) DecodedLen(n int) int
```

DecodedLen returns the maximum length in bytes of the decoded data

corresponding to n bytes of base64-encoded data.

func (*Encoding) [Encode](#)

```
func (enc *Encoding) Encode(dst, src []byte)
```

Encode encodes src using the encoding enc, writing EncodedLen(len(src)) bytes to dst.

The encoding pads the output to a multiple of 4 bytes, so Encode is not appropriate for use on individual blocks of a large data stream. Use NewEncoder() instead.

func (*Encoding) [EncodeToString](#)

```
func (enc *Encoding) EncodeToString(src []byte) string
```

EncodeToString returns the base64 encoding of src.

func (*Encoding) [EncodedLen](#)

```
func (enc *Encoding) EncodedLen(n int) int
```

EncodedLen returns the length in bytes of the base64 encoding of an input buffer of length n.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package binary

```
import "encoding/binary"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package `binary` implements translation between numbers and byte sequences and encoding and decoding of varints.

Numbers are translated by reading and writing fixed-size values. A fixed-size value is either a fixed-size arithmetic type (`int8`, `uint8`, `int16`, `float32`, `complex64`, ...) or an array or struct containing only fixed-size values.

Varints are a method of encoding integers using one or more bytes; numbers with smaller absolute value take a smaller number of bytes. For a specification, see <http://code.google.com/apis/protocolbuffers/docs/encoding.html>.

Index

[Constants](#)

[Variables](#)

[func PutUvarint\(buf \[\]byte, x uint64\) int](#)

[func PutVarint\(buf \[\]byte, x int64\) int](#)

[func Read\(r io.Reader, order ByteOrder, data interface{ }\) error](#)

[func ReadUvarint\(r io.ByteReader\) \(uint64, error\)](#)

[func ReadVarint\(r io.ByteReader\) \(int64, error\)](#)

[func Size\(v interface{ }\) int](#)

[func Uvarint\(buf \[\]byte\) \(uint64, int\)](#)

[func Varint\(buf \[\]byte\) \(int64, int\)](#)

[func Write\(w io.Writer, order ByteOrder, data interface{ }\) error](#)

[type ByteOrder](#)

Examples

[Read](#)

[Write](#)

[Write \(Multi\)](#)

Package files

[binary.go](#) [varint.go](#)

Constants

```
const (  
    MaxVarintLen16 = 3  
    MaxVarintLen32 = 5  
    MaxVarintLen64 = 10  
)
```

MaxVarintLenN is the maximum length of a varint-encoded N-bit integer.

Variables

```
var BigEndian bigEndian
```

BigEndian is the big-endian implementation of ByteOrder.

```
var LittleEndian littleEndian
```

LittleEndian is the little-endian implementation of ByteOrder.

func [PutUvarint](#)

```
func PutUvarint(buf []byte, x uint64) int
```

PutUvarint encodes a uint64 into buf and returns the number of bytes written. If the buffer is too small, PutUvarint will panic.

func [PutVarint](#)

```
func PutVarint(buf []byte, x int64) int
```

PutVarint encodes an int64 into buf and returns the number of bytes written. If the buffer is too small, PutVarint will panic.

func [Read](#)

```
func Read(r io.Reader, order ByteOrder, data interface{}) error
```

Read reads structured binary data from r into data. Data must be a pointer to a fixed-size value or a slice of fixed-size values. Bytes read from r are decoded using the specified byte order and written to successive fields of the data.

? Example

? Example

Code:

```
var pi float64
b := []byte{0x18, 0x2d, 0x44, 0x54, 0xfb, 0x21, 0x09, 0x40}
buf := bytes.NewBuffer(b)
err := binary.Read(buf, binary.LittleEndian, &pi)
if err != nil {
    fmt.Println("binary.Read failed:", err)
}
fmt.Print(pi)
```

Output:

```
3.141592653589793
```

func [ReadUvarint](#)

```
func ReadUvarint(r io.ByteReader) (uint64, error)
```

ReadUvarint reads an encoded unsigned integer from r and returns it as a uint64.

func [ReadVarint](#)

```
func ReadVarint(r io.ByteReader) (int64, error)
```

ReadVarint reads an encoded unsigned integer from r and returns it as a uint64.

func [Size](#)

```
func Size(v interface{}) int
```

Size returns how many bytes Write would generate to encode the value v, which must be a fixed-size value or a slice of fixed-size values, or a pointer to such data.

func [Uvarint](#)

```
func Uvarint(buf []byte) (uint64, int)
```

Uvarint decodes a uint64 from buf and returns that value and the number of bytes read (> 0). If an error occurred, the value is 0 and the number of bytes n is <= 0 meaning:

```
n == 0: buf too small  
n < 0: value larger than 64 bits (overflow)  
       and -n is the number of bytes read
```

func [Varint](#)

```
func Varint(buf []byte) (int64, int)
```

Varint decodes an int64 from buf and returns that value and the number of bytes read (> 0). If an error occurred, the value is 0 and the number of bytes n is <= 0 with the following meaning:

```
n == 0: buf too small  
n < 0: value larger than 64 bits (overflow)  
       and -n is the number of bytes read
```

func [Write](#)

```
func Write(w io.Writer, order ByteOrder, data interface{}) error
```

Write writes the binary representation of data into w. Data must be a fixed-size value or a slice of fixed-size values, or a pointer to such data. Bytes written to w are encoded using the specified byte order and read from successive fields of the data.

? Example

? Example

Code:

```
buf := new(bytes.Buffer)
var pi float64 = math.Pi
err := binary.Write(buf, binary.LittleEndian, pi)
if err != nil {
    fmt.Println("binary.Write failed:", err)
}
fmt.Printf("% x", buf.Bytes())
```

Output:

```
18 2d 44 54 fb 21 09 40
```

? Example (Multi)

? Example (Multi)

Code:

```
buf := new(bytes.Buffer)
var data = []interface{}{
    uint16(61374),
    int8(-54),
    uint8(254),
}
for _, v := range data {
    err := binary.Write(buf, binary.LittleEndian, v)
    if err != nil {
        fmt.Println("binary.Write failed:", err)
    }
}
```

```
    }  
}  
fmt.Printf("%x", buf.Bytes())
```

Output:

beefcafe

type [ByteOrder](#)

```
type ByteOrder interface {
    Uint16([]byte) uint16
    Uint32([]byte) uint32
    Uint64([]byte) uint64
    PutUint16([]byte, uint16)
    PutUint32([]byte, uint32)
    PutUint64([]byte, uint64)
    String() string
}
```

A `ByteOrder` specifies how to convert byte sequences into 16-, 32-, or 64-bit unsigned integers.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package csv

```
import "encoding/csv"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package csv reads and writes comma-separated values (CSV) files.

A csv file contains zero or more records of one or more fields per record. Each record is separated by the newline character. The final record may optionally be followed by a newline character.

```
field1,field2,field3
```

White space is considered part of a field.

Carriage returns before newline characters are silently removed.

Blank lines are ignored. A line with only whitespace characters (excluding the ending newline character) is not considered a blank line.

Fields which start and stop with the quote character " are called quoted-fields. The beginning and ending quote are not part of the field.

The source:

```
normal string,"quoted-field"
```

results in the fields

```
{`normal string`, `quoted-field`}
```

Within a quoted-field a quote character followed by a second quote character is considered a single quote.

```
"the ""word"" is true","a ""quoted-field"""
```

results in

```
{`the "word" is true`, `a "quoted-field"`}
```

Newlines and commas may be included in a quoted-field

```
"Multi-line  
field","comma is ,"
```

results in

```
{`Multi-line  
field`, `comma is ,`}
```

Index

Variables

type ParseError

func (e *ParseError) Error() string

type Reader

func NewReader(r io.Reader) *Reader

func (r *Reader) Read() (record []string, err error)

func (r *Reader) ReadAll() (records [][]string, err error)

type Writer

func NewWriter(w io.Writer) *Writer

func (w *Writer) Flush()

func (w *Writer) Write(record []string) (err error)

func (w *Writer) WriteAll(records [][]string) (err error)

Package files

[reader.go](#) [writer.go](#)

Variables

```
var (  
    ErrTrailingComma = errors.New("extra delimiter at end of line")  
    ErrBareQuote     = errors.New("bare \" in non-quoted-field")  
    ErrQuote         = errors.New("extraneous \" in field")  
    ErrFieldCount    = errors.New("wrong number of fields in line")  
)
```

These are the errors that can be returned in `ParseError.Error`

type [ParseError](#)

```
type ParseError struct {  
    Line    int    // Line where the error occurred  
    Column  int    // Column (rune index) where the error occurred  
    Err     error  // The actual error  
}
```

A ParseError is returned for parsing errors. The first line is 1. The first column is 0.

func (*ParseError) [Error](#)

```
func (e *ParseError) Error() string
```

type [Reader](#)

```
type Reader struct {
    Comma          rune // Field delimiter (set to ',' by NewReader)
    Comment        rune // Comment character for start of line
    FieldsPerRecord int  // Number of expected fields per record
    LazyQuotes     bool // Allow lazy quotes
    TrailingComma  bool // Allow trailing comma
    TrimLeadingSpace bool // Trim leading space
    // contains filtered or unexported fields
}
```

A Reader reads records from a CSV-encoded file.

As returned by `NewReader`, a Reader expects input conforming to RFC 4180. The exported fields can be changed to customize the details before the first call to `Read` or `ReadAll`.

`Comma` is the field delimiter. It defaults to `'`.

`Comment`, if not 0, is the comment character. Lines beginning with the `Comment` character are ignored.

If `FieldsPerRecord` is positive, `Read` requires each record to have the given number of fields. If `FieldsPerRecord` is 0, `Read` sets it to the number of fields in the first record, so that future records must have the same field count. If `FieldsPerRecord` is negative, no check is made and records may have a variable number of fields.

If `LazyQuotes` is true, a quote may appear in an unquoted field and a non-doubled quote may appear in a quoted field.

If `TrailingComma` is true, the last field may be an unquoted empty field.

If `TrimLeadingSpace` is true, leading white space in a field is ignored.

func [NewReader](#)

```
func NewReader(r io.Reader) *Reader
```

NewReader returns a new Reader that reads from r.

func (*Reader) [Read](#)

```
func (r *Reader) Read() (record []string, err error)
```

Read reads one record from r. The record is a slice of strings with each string representing one field.

func (*Reader) [ReadAll](#)

```
func (r *Reader) ReadAll() (records [][]string, err error)
```

ReadAll reads all the remaining records from r. Each record is a slice of fields. A successful call returns `err == nil`, not `err == EOF`. Because ReadAll is defined to read until EOF, it does not treat end of file as an error to be reported.

type [Writer](#)

```
type Writer struct {  
    Comma    rune // Field delimiter (set to to ',' by NewWriter)  
    UseCRLF  bool // True to use \r\n as the line terminator  
    // contains filtered or unexported fields  
}
```

A Writer writes records to a CSV encoded file.

As returned by `NewWriter`, a Writer writes records terminated by a newline and uses ',' as the field delimiter. The exported fields can be changed to customize the details before the first call to `Write` or `WriteAll`.

Comma is the field delimiter.

If `UseCRLF` is true, the Writer ends each record with `\r\n` instead of `\n`.

func [NewWriter](#)

```
func NewWriter(w io.Writer) *Writer
```

`NewWriter` returns a new `Writer` that writes to `w`.

func (*Writer) [Flush](#)

```
func (w *Writer) Flush()
```

`Flush` writes any buffered data to the underlying `io.Writer`.

func (*Writer) [Write](#)

```
func (w *Writer) Write(record []string) (err error)
```

`Writer` writes a single CSV record to `w` along with any necessary quoting. A record is a slice of strings with each string being one field.

func (*Writer) [WriteAll](#)

```
func (w *Writer) WriteAll(records [][]string) (err error)
```

WriteAll writes multiple CSV records to w using Write and then calls Flush.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package gob

```
import "encoding/gob"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `gob` manages streams of gobs - binary values exchanged between an Encoder (transmitter) and a Decoder (receiver). A typical use is transporting arguments and results of remote procedure calls (RPCs) such as those provided by package `"rpc"`.

A stream of gobs is self-describing. Each data item in the stream is preceded by a specification of its type, expressed in terms of a small set of predefined types. Pointers are not transmitted, but the things they point to are transmitted; that is, the values are flattened. Recursive types work fine, but recursive values (data with cycles) are problematic. This may change.

To use gobs, create an Encoder and present it with a series of data items as values or addresses that can be dereferenced to values. The Encoder makes sure all type information is sent before it is needed. At the receive side, a Decoder retrieves values from the encoded stream and unpacks them into local variables.

The source and destination values/types need not correspond exactly. For structs, fields (identified by name) that are in the source but absent from the receiving variable will be ignored. Fields that are in the receiving variable but missing from the transmitted type or value will be ignored in the destination. If a field with the same name is present in both, their types must be compatible. Both the receiver and transmitter will do all necessary indirection and dereferencing to convert between gobs and actual Go values. For instance, a gob type that is schematically,

```
struct { A, B int }
```

can be sent from or received into any of these Go types:

```
struct { A, B int }      // the same
*struct { A, B int }    // extra indirection of the struct
struct { *A, **B int }  // extra indirection of the fields
struct { A, B int64 }   // different concrete value type; see below
```

It may also be received into any of these:

```
struct { A, B int }      // the same
struct { B, A int }      // ordering doesn't matter; matching is by n
```

```
struct { A, B, C int } // extra field (C) ignored
struct { B int } // missing field (A) ignored; data will be d
struct { B, C int } // missing field (A) ignored; extra field (C
```

Attempting to receive into these types will draw a decode error:

```
struct { A int; B uint } // change of signedness for B
struct { A int; B float } // change of type for B
struct { } // no field names in common
struct { C, D int } // no field names in common
```

Integers are transmitted two ways: arbitrary precision signed integers or arbitrary precision unsigned integers. There is no int8, int16 etc. discrimination in the gob format; there are only signed and unsigned integers. As described below, the transmitter sends the value in a variable-length encoding; the receiver accepts the value and stores it in the destination variable. Floating-point numbers are always sent using IEEE-754 64-bit precision (see below).

Signed integers may be received into any signed integer variable: int, int16, etc.; unsigned integers may be received into any unsigned integer variable; and floating point values may be received into any floating point variable. However, the destination variable must be able to represent the value or the decode operation will fail.

Structs, arrays and slices are also supported. Strings and arrays of bytes are supported with a special, efficient representation (see below). When a slice is decoded, if the existing slice has capacity the slice will be extended in place; if not, a new array is allocated. Regardless, the length of the resulting slice reports the number of elements decoded.

Functions and channels cannot be sent in a gob. Attempting to encode a value that contains one will fail.

The rest of this comment documents the encoding, details that are not important for most users. Details are presented bottom-up.

An unsigned integer is sent one of two ways. If it is less than 128, it is sent as a byte with that value. Otherwise it is sent as a minimal-length big-endian (high byte first) byte stream holding the value, preceded by one byte holding the byte count, negated. Thus 0 is transmitted as (00), 7 is transmitted as (07) and 256 is transmitted as (FE 01 00).

A boolean is encoded within an unsigned integer: 0 for false, 1 for true.

A signed integer, i , is encoded within an unsigned integer, u . Within u , bits 1 upward contain the value; bit 0 says whether they should be complemented upon receipt. The encode algorithm looks like this:

```
uint u;
if i < 0 {
    u = (^i << 1) | 1    // complement i, bit 0 is 1
} else {
    u = (i << 1)    // do not complement i, bit 0 is 0
}
encodeUnsigned(u)
```

The low bit is therefore analogous to a sign bit, but making it the complement bit instead guarantees that the largest negative integer is not a special case. For example, $-129 = ^{128} = (^{256} >> 1)$ encodes as (FE 01 01).

Floating-point numbers are always sent as a representation of a float64 value. That value is converted to a uint64 using `math.Float64bits`. The uint64 is then byte-reversed and sent as a regular unsigned integer. The byte-reversal means the exponent and high-precision part of the mantissa go first. Since the low bits are often zero, this can save encoding bytes. For instance, 17.0 is encoded in only three bytes (FE 31 40).

Strings and slices of bytes are sent as an unsigned count followed by that many uninterpreted bytes of the value.

All other slices and arrays are sent as an unsigned count followed by that many elements using the standard gob encoding for their type, recursively.

Maps are sent as an unsigned count followed by that many key, element pairs. Empty but non-nil maps are sent, so if the sender has allocated a map, the receiver will allocate a map even no elements are transmitted.

Structs are sent as a sequence of (field number, field value) pairs. The field value is sent using the standard gob encoding for its type, recursively. If a field has the zero value for its type, it is omitted from the transmission. The field number is defined by the type of the encoded struct: the first field of the encoded type is field 0, the second is field 1, etc. When encoding a value, the field numbers are delta encoded for efficiency and the fields are always sent in order of increasing

field number; the deltas are therefore unsigned. The initialization for the delta encoding sets the field number to -1, so an unsigned integer field 0 with value 7 is transmitted as unsigned delta = 1, unsigned value = 7 or (01 07). Finally, after all the fields have been sent a terminating mark denotes the end of the struct. That mark is a delta=0 value, which has representation (00).

Interface types are not checked for compatibility; all interface types are treated, for transmission, as members of a single "interface" type, analogous to int or []byte - in effect they're all treated as interface{}. Interface values are transmitted as a string identifying the concrete type being sent (a name that must be pre-defined by calling Register), followed by a byte count of the length of the following data (so the value can be skipped if it cannot be stored), followed by the usual encoding of concrete (dynamic) value stored in the interface value. (A nil interface value is identified by the empty string and transmits no value.) Upon receipt, the decoder verifies that the unpacked concrete item satisfies the interface of the receiving variable.

The representation of types is described below. When a type is defined on a given connection between an Encoder and Decoder, it is assigned a signed integer type id. When Encoder.Encode(v) is called, it makes sure there is an id assigned for the type of v and all its elements and then it sends the pair (typeid, encoded-v) where typeid is the type id of the encoded type of v and encoded-v is the gob encoding of the value v.

To define a type, the encoder chooses an unused, positive type id and sends the pair (-type id, encoded-type) where encoded-type is the gob encoding of a wireType description, constructed from these types:

```
type wireType struct {
    ArrayT  *ArrayType
    SliceT  *SliceType
    StructT *StructType
    MapT    *MapType
}
type arrayType struct {
    CommonType
    Elem typeId
    Len  int
}
type CommonType struct {
    Name string // the name of the struct type
    Id   int     // the id of the type, repeated so it's inside th
```

```

}
type sliceType struct {
    CommonType
    Elem typeId
}
type structType struct {
    CommonType
    Field []*fieldType // the fields of the struct.
}
type fieldType struct {
    Name string // the name of the field.
    Id    int    // the type id of the field, which must be alrea
}
type mapType struct {
    CommonType
    Key  typeId
    Elem typeId
}
}

```

If there are nested type ids, the types for all inner type ids must be defined before the top-level type id is used to describe an encoded-v.

For simplicity in setup, the connection is defined to understand these types a priori, as well as the basic gob types int, uint, etc. Their ids are:

```

bool        1
int         2
uint        3
float       4
[]byte      5
string      6
complex     7
interface   8
// gap for reserved ids.
WireType    16
ArrayType    17
CommonType  18
SliceType   19
StructType  20
FieldType   21
// 22 is slice of fieldType.
MapType     23

```

Finally, each message created by a call to Encode is preceded by an encoded unsigned integer count of the number of bytes remaining in the message. After the initial type name, interface values are wrapped the same way; in effect, the interface value acts like a recursive invocation of Encode.

In summary, a gob stream looks like

`(byteCount (-type id, encoding of a wireType))* (type id, encoding of`

where * signifies zero or more repetitions and the type id of a value must be predefined or be defined before the value in the stream.

See "Gobs of data" for a design discussion of the gob wire format:

http://golang.org/doc/articles/gobs_of_data.html

Index

[func Register\(value interface{}\)](#)

[func RegisterName\(name string, value interface{}\)](#)

[type CommonType](#)

[type Decoder](#)

[func NewDecoder\(r io.Reader\) *Decoder](#)

[func \(dec *Decoder\) Decode\(e interface{}\) error](#)

[func \(dec *Decoder\) DecodeValue\(v reflect.Value\) error](#)

[type Encoder](#)

[func NewEncoder\(w io.Writer\) *Encoder](#)

[func \(enc *Encoder\) Encode\(e interface{}\) error](#)

[func \(enc *Encoder\) EncodeValue\(value reflect.Value\) error](#)

[type GobDecoder](#)

[type GobEncoder](#)

Package files

[decode.go](#) [decoder.go](#) [doc.go](#) [encode.go](#) [encoder.go](#) [error.go](#) [type.go](#)

func Register

```
func Register(value interface{})
```

Register records a type, identified by a value for that type, under its internal type name. That name will identify the concrete type of a value sent or received as an interface variable. Only types that will be transferred as implementations of interface values need to be registered. Expecting to be used only during initialization, it panics if the mapping between types and names is not a bijection.

func [RegisterName](#)

```
func RegisterName(name string, value interface{})
```

RegisterName is like Register but uses the provided name rather than the type's default.

type [CommonType](#)

```
type CommonType struct {  
    Name string  
    Id    typeId  
}
```

CommonType holds elements of all types. It is a historical artifact, kept for binary compatibility and exported only for the benefit of the package's encoding of type descriptors. It is not intended for direct use by clients.

type [Decoder](#)

```
type Decoder struct {  
    // contains filtered or unexported fields  
}
```

A Decoder manages the receipt of type and data information read from the remote side of a connection.

func [NewDecoder](#)

```
func NewDecoder(r io.Reader) *Decoder
```

NewDecoder returns a new decoder that reads from the io.Reader. If r does not also implement io.ByteReader, it will be wrapped in a bufio.Reader.

func (***Decoder**) [Decode](#)

```
func (dec *Decoder) Decode(e interface{}) error
```

Decode reads the next value from the connection and stores it in the data represented by the empty interface value. If e is nil, the value will be discarded. Otherwise, the value underlying e must be a pointer to the correct type for the next data item received.

func (***Decoder**) [DecodeValue](#)

```
func (dec *Decoder) DecodeValue(v reflect.Value) error
```

DecodeValue reads the next value from the connection. If v is the zero reflect.Value (v.Kind() == Invalid), DecodeValue discards the value. Otherwise, it stores the value into v. In that case, v must represent a non-nil pointer to data or be an assignable reflect.Value (v.CanSet())

type [Encoder](#)

```
type Encoder struct {  
    // contains filtered or unexported fields  
}
```

An Encoder manages the transmission of type and data information to the other side of a connection.

func [NewEncoder](#)

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder returns a new encoder that will transmit on the io.Writer.

func (***Encoder**) [Encode](#)

```
func (enc *Encoder) Encode(e interface{}) error
```

Encode transmits the data item represented by the empty interface value, guaranteeing that all necessary type information has been transmitted first.

func (***Encoder**) [EncodeValue](#)

```
func (enc *Encoder) EncodeValue(value reflect.Value) error
```

EncodeValue transmits the data item represented by the reflection value, guaranteeing that all necessary type information has been transmitted first.

type [GobDecoder](#)

```
type GobDecoder interface {  
    // GobDecode overwrites the receiver, which must be a pointer,  
    // with the value represented by the byte slice, which was writt  
    // by GobEncode, usually for the same concrete type.  
    GobDecode([]byte) error  
}
```

GobDecoder is the interface describing data that provides its own routine for decoding transmitted values sent by a GobEncoder.

type [GobEncoder](#)

```
type GobEncoder interface {  
    // GobEncode returns a byte slice representing the encoding of t  
    // receiver for transmission to a GobDecoder, usually of the same  
    // concrete type.  
    GobEncode() ([]byte, error)  
}
```

GobEncoder is the interface describing data that provides its own representation for encoding values for transmission to a GobDecoder. A type that implements GobEncoder and GobDecoder has complete control over the representation of its data and may therefore contain things such as private fields, channels, and functions, which are not usually transmissible in gob streams.

Note: Since gobs can be stored permanently, It is good design to guarantee the encoding used by a GobEncoder is stable as the software evolves. For instance, it might make sense for GobEncode to include a version number in the encoding.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package hex

```
import "encoding/hex"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package hex implements hexadecimal encoding and decoding.

Index

Variables

func Decode(dst, src []byte) (int, error)
func DecodeString(s string) ([]byte, error)
func DecodedLen(x int) int
func Dump(data []byte) string
func Dumper(w io.Writer) io.WriteCloser
func Encode(dst, src []byte) int
func EncodeToString(src []byte) string
func EncodedLen(n int) int
type InvalidByteError
func (e InvalidByteError) Error() string

Package files

hex.go

Variables

```
var ErrLength = errors.New("encoding/hex: odd length hex string")
```

ErrLength results from decoding an odd length slice.

func [Decode](#)

```
func Decode(dst, src []byte) (int, error)
```

Decode decodes src into DecodedLen(len(src)) bytes, returning the actual number of bytes written to dst.

If Decode encounters invalid input, it returns an error describing the failure.

func DecodeString

```
func DecodeString(s string) ([]byte, error)
```

DecodeString returns the bytes represented by the hexadecimal string s.

func DecodedLen

```
func DecodedLen(x int) int
```

func [Dump](#)

```
func Dump(data []byte) string
```

Dump returns a string that contains a hex dump of the given data. The format of the hex dump matches the output of `hexdump -C` on the command line.

func [Dumper](#)

```
func Dumper(w io.Writer) io.WriteCloser
```

Dumper returns a WriteCloser that writes a hex dump of all written data to w. The format of the dump matches the output of `hexdump -C` on the command line.

func [Encode](#)

```
func Encode(dst, src []byte) int
```

Encode encodes src into EncodedLen(len(src)) bytes of dst. As a convenience, it returns the number of bytes written to dst, but this value is always EncodedLen(len(src)). Encode implements hexadecimal encoding.

func [EncodeToString](#)

```
func EncodeToString(src []byte) string
```

EncodeToString returns the hexadecimal encoding of src.

func EncodedLen

```
func EncodedLen(n int) int
```

EncodedLen returns the length of an encoding of n source bytes.

type [InvalidByteError](#)

```
type InvalidByteError byte
```

InvalidByteError values describe errors resulting from an invalid byte in a hex string.

func (InvalidByteError) [Error](#)

```
func (e InvalidByteError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package json

```
import "encoding/json"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package json implements encoding and decoding of JSON objects as defined in RFC 4627.

See "JSON and Go" for an introduction to this package:

http://golang.org/doc/articles/json_and_go.html

Index

[func Compact\(dst *bytes.Buffer, src \[\]byte\) error](#)
[func HTMLEscape\(dst *bytes.Buffer, src \[\]byte\)](#)
[func Indent\(dst *bytes.Buffer, src \[\]byte, prefix, indent string\) error](#)
[func Marshal\(v interface{}\) \(\[\]byte, error\)](#)
[func MarshalIndent\(v interface{}, prefix, indent string\) \(\[\]byte, error\)](#)
[func Unmarshal\(data \[\]byte, v interface{}\) error](#)
[type Decoder](#)
[func NewDecoder\(r io.Reader\) *Decoder](#)
[func \(dec *Decoder\) Decode\(v interface{}\) error](#)
[type Encoder](#)
[func NewEncoder\(w io.Writer\) *Encoder](#)
[func \(enc *Encoder\) Encode\(v interface{}\) error](#)
[type InvalidUTF8Error](#)
[func \(e *InvalidUTF8Error\) Error\(\) string](#)
[type InvalidUnmarshalError](#)
[func \(e *InvalidUnmarshalError\) Error\(\) string](#)
[type Marshaler](#)
[type MarshalerError](#)
[func \(e *MarshalerError\) Error\(\) string](#)
[type RawMessage](#)
[func \(m *RawMessage\) MarshalJSON\(\) \(\[\]byte, error\)](#)
[func \(m *RawMessage\) UnmarshalJSON\(data \[\]byte\) error](#)
[type SyntaxError](#)
[func \(e *SyntaxError\) Error\(\) string](#)
[type UnmarshalFieldError](#)
[func \(e *UnmarshalFieldError\) Error\(\) string](#)
[type UnmarshalTypeError](#)
[func \(e *UnmarshalTypeError\) Error\(\) string](#)
[type Unmarshaler](#)
[type UnsupportedTypeError](#)
[func \(e *UnsupportedTypeError\) Error\(\) string](#)
[type UnsupportedValueError](#)
[func \(e *UnsupportedValueError\) Error\(\) string](#)
[Bugs](#)

Examples

[Decoder](#)

[Marshal](#)

[Unmarshal](#)

Package files

[decode.go](#) [encode.go](#) [indent.go](#) [scanner.go](#) [stream.go](#) [tags.go](#)

func Compact

```
func Compact(dst *bytes.Buffer, src []byte) error
```

Compact appends to dst the JSON-encoded src with insignificant space characters elided.

func [HTMLEscape](#)

```
func HTMLEscape(dst *bytes.Buffer, src []byte)
```

HTMLEscape appends to dst the JSON-encoded src with <, >, and & characters inside string literals changed to \u003c, \u003e, \u0026 so that the JSON will be safe to embed inside HTML <script> tags. For historical reasons, web browsers don't honor standard HTML escaping within <script> tags, so an alternative JSON encoding must be used.

func Indent

```
func Indent(dst *bytes.Buffer, src []byte, prefix, indent string) error
```

Indent appends to dst an indented form of the JSON-encoded src. Each element in a JSON object or array begins on a new, indented line beginning with prefix followed by one or more copies of indent according to the indentation nesting. The data appended to dst has no trailing newline, to make it easier to embed inside other formatted JSON data.

func [Marshal](#)

```
func Marshal(v interface{}) ([]byte, error)
```

Marshal returns the JSON encoding of v.

Marshal traverses the value v recursively. If an encountered value implements the Marshaler interface and is not a nil pointer, Marshal calls its MarshalJSON method to produce JSON. The nil pointer exception is not strictly necessary but mimics a similar, necessary exception in the behavior of UnmarshalJSON.

Otherwise, Marshal uses the following type-dependent default encodings:

Boolean values encode as JSON booleans.

Floating point and integer values encode as JSON numbers.

String values encode as JSON strings, with each invalid UTF-8 sequence replaced by the encoding of the Unicode replacement character U+FFFD. The angle brackets "<" and ">" are escaped to "\u003c" and "\u003e" to keep some browsers from misinterpreting JSON output as HTML.

Array and slice values encode as JSON arrays, except that []byte encodes as a base64-encoded string, and a nil slice encodes as the null JSON object.

Struct values encode as JSON objects. Each exported struct field becomes a member of the object unless

- the field's tag is "-", or
- the field is empty and its tag specifies the "omitempty" option.

The empty values are false, 0, any nil pointer or interface value, and any array, slice, map, or string of length zero. The object's default key string is the struct field name but can be specified in the struct field's tag value. The "json" key in struct field's tag value is the key name, followed by an optional comma and options. Examples:

```
// Field is ignored by this package.  
Field int `json:"- "`
```

```
// Field appears in JSON as key "myName".
Field int `json:"myName"`

// Field appears in JSON as key "myName" and
// the field is omitted from the object if its value is empty,
// as defined above.
Field int `json:"myName,omitempty"`

// Field appears in JSON as key "Field" (the default), but
// the field is skipped if empty.
// Note the leading comma.
Field int `json:",omitempty"`
```

The "string" option signals that a field is stored as JSON inside a JSON-encoded string. This extra level of encoding is sometimes used when communicating with JavaScript programs:

```
Int64String int64 `json:",string"`
```

The key name will be used if it's a non-empty string consisting of only Unicode letters, digits, dollar signs, percent signs, hyphens, underscores and slashes.

Map values encode as JSON objects. The map's key type must be string; the object keys are used directly as map keys.

Pointer values encode as the value pointed to. A nil pointer encodes as the null JSON object.

Interface values encode as the value contained in the interface. A nil interface value encodes as the null JSON object.

Channel, complex, and function values cannot be encoded in JSON. Attempting to encode such a value causes Marshal to return an `InvalidTypeError`.

JSON cannot represent cyclic data structures and Marshal does not handle them. Passing cyclic structures to Marshal will result in an infinite recursion.

? Example

? Example

Code:

```
type ColorGroup struct {
    ID      int
    Name    string
    Colors []string
}
group := ColorGroup{
    ID:      1,
    Name:    "Reds",
    Colors: []string{"Crimson", "Red", "Ruby", "Maroon"},
}
b, err := json.Marshal(group)
if err != nil {
    fmt.Println("error:", err)
}
os.Stdout.Write(b)
```

Output:

```
{"ID":1,"Name":"Reds","Colors":["Crimson","Red","Ruby","Maroon"]}
```

func [MarshalIndent](#)

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

MarshalIndent is like Marshal but applies Indent to format the output.

func [Unmarshal](#)

```
func Unmarshal(data []byte, v interface{}) error
```

Unmarshal parses the JSON-encoded data and stores the result in the value pointed to by v.

Unmarshal uses the inverse of the encodings that Marshal uses, allocating maps, slices, and pointers as necessary, with the following additional rules:

To unmarshal JSON into a pointer, Unmarshal first handles the case of the JSON being the JSON literal null. In that case, Unmarshal sets the pointer to nil. Otherwise, Unmarshal unmarshals the JSON into the value pointed at by the pointer. If the pointer is nil, Unmarshal allocates a new value for it to point to.

To unmarshal JSON into an interface value, Unmarshal unmarshals the JSON into the concrete value contained in the interface value. If the interface value is nil, that is, has no concrete value stored in it, Unmarshal stores one of these in the interface value:

```
bool, for JSON booleans
float64, for JSON numbers
string, for JSON strings
[]interface{}, for JSON arrays
map[string]interface{}, for JSON objects
nil for JSON null
```

If a JSON value is not appropriate for a given target type, or if a JSON number overflows the target type, Unmarshal skips that field and completes the unmarshalling as best it can. If no more serious errors are encountered, Unmarshal returns an `UnmarshalTypeError` describing the earliest such error.

? Example

? Example

Code:

```
var jsonBlob = []byte(`[
  {"Name": "Platypus", "Order": "Monotremata"},
  {"Name": "Quoll", "Order": "Dasyuromorphia"}
]`)
```

```
]`)  
type Animal struct {  
    Name string  
    Order string  
}  
var animals []Animal  
err := json.Unmarshal(jsonBlob, &animals)  
if err != nil {  
    fmt.Println("error:", err)  
}  
fmt.Printf("%+v", animals)
```

Output:

```
[{Name:Platypus Order:Monotremata} {Name:Quoll Order:Dasyuromorphia}]
```

type [Decoder](#)

```
type Decoder struct {  
    // contains filtered or unexported fields  
}
```

A Decoder reads and decodes JSON objects from an input stream.

? Example

? Example

This example uses a Decoder to decode a stream of distinct JSON values.

Code:

```
const jsonStream = `  
    {"Name": "Ed", "Text": "Knock knock."}  
    {"Name": "Sam", "Text": "Who's there?"}  
    {"Name": "Ed", "Text": "Go fmt."}  
    {"Name": "Sam", "Text": "Go fmt who?"}  
    {"Name": "Ed", "Text": "Go fmt yourself!"}  
`  
  
type Message struct {  
    Name, Text string  
}  
dec := json.NewDecoder(strings.NewReader(jsonStream))  
for {  
    var m Message  
    if err := dec.Decode(&m); err == io.EOF {  
        break  
    } else if err != nil {  
        log.Fatal(err)  
    }  
    fmt.Printf("%s: %s\n", m.Name, m.Text)  
}
```

Output:

```
Ed: Knock knock.  
Sam: Who's there?  
Ed: Go fmt.  
Sam: Go fmt who?  
Ed: Go fmt yourself!
```

func [NewDecoder](#)

```
func NewDecoder(r io.Reader) *Decoder
```

NewDecoder returns a new decoder that reads from r.

The decoder introduces its own buffering and may read data from r beyond the JSON values requested.

func (*Decoder) [Decode](#)

```
func (dec *Decoder) Decode(v interface{}) error
```

Decode reads the next JSON-encoded value from its input and stores it in the value pointed to by v.

See the documentation for Unmarshal for details about the conversion of JSON into a Go value.

type [Encoder](#)

```
type Encoder struct {  
    // contains filtered or unexported fields  
}
```

An Encoder writes JSON objects to an output stream.

func [NewEncoder](#)

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder returns a new encoder that writes to w.

func (***Encoder**) [Encode](#)

```
func (enc *Encoder) Encode(v interface{}) error
```

Encode writes the JSON encoding of v to the connection.

See the documentation for Marshal for details about the conversion of Go values to JSON.

type [InvalidUTF8Error](#)

```
type InvalidUTF8Error struct {  
    S string  
}
```

func ([*InvalidUTF8Error](#)) [Error](#)

```
func (e \*InvalidUTF8Error) Error() string
```

type [InvalidUnmarshalError](#)

```
type InvalidUnmarshalError struct {  
    Type reflect.Type  
}
```

An `InvalidUnmarshalError` describes an invalid argument passed to `Unmarshal`. (The argument to `Unmarshal` must be a non-nil pointer.)

func ([*InvalidUnmarshalError](#)) [Error](#)

```
func (e \*InvalidUnmarshalError) Error() string
```

type [Marshaler](#)

```
type Marshaler interface {  
    MarshalJSON() ([]byte, error)  
}
```

Marshaler is the interface implemented by objects that can marshal themselves into valid JSON.

type [MarshalerError](#)

```
type MarshalerError struct {  
    Type reflect.Type  
    Err  error  
}
```

func (*MarshalerError) [Error](#)

```
func (e *MarshalerError) Error() string
```

type [RawMessage](#)

```
type RawMessage []byte
```

RawMessage is a raw encoded JSON object. It implements Marshaler and Unmarshaler and can be used to delay JSON decoding or precompute a JSON encoding.

func (*RawMessage) [MarshalJSON](#)

```
func (m *RawMessage) MarshalJSON() ([]byte, error)
```

MarshalJSON returns *m as the JSON encoding of m.

func (*RawMessage) [UnmarshalJSON](#)

```
func (m *RawMessage) UnmarshalJSON(data []byte) error
```

UnmarshalJSON sets *m to a copy of data.

type [SyntaxError](#)

```
type SyntaxError struct {  
    Offset int64 // error occurred after reading Offset bytes  
    // contains filtered or unexported fields  
}
```

A `SyntaxError` is a description of a JSON syntax error.

func ([*SyntaxError](#)) [Error](#)

```
func (e *SyntaxError) Error() string
```

type [UnmarshalFieldError](#)

```
type UnmarshalFieldError struct {  
    Key    string  
    Type   reflect.Type  
    Field  reflect.StructField  
}
```

An `UnmarshalFieldError` describes a JSON object key that led to an unexported (and therefore unwritable) struct field.

func ([*UnmarshalFieldError](#)) [Error](#)

```
func (e *UnmarshalFieldError) Error() string
```

type [UnmarshalTypeError](#)

```
type UnmarshalTypeError struct {  
    Value string // description of JSON value - "bool", "array"  
    Type  reflect.Type // type of Go value it could not be assigned  
}
```

An `UnmarshalTypeError` describes a JSON value that was not appropriate for a value of a specific Go type.

func ([*UnmarshalTypeError](#)) [Error](#)

```
func (e *UnmarshalTypeError) Error() string
```

type Unmarshaler

```
type Unmarshaler interface {  
    UnmarshalJSON([]byte) error  
}
```

Unmarshaler is the interface implemented by objects that can unmarshal a JSON description of themselves. The input can be assumed to be a valid JSON object encoding. UnmarshalJSON must copy the JSON data if it wishes to retain the data after returning.

type [UnsupportedTypeError](#)

```
type UnsupportedTypeError struct {  
    Type reflect.Type  
}
```

func (*UnsupportedTypeError) [Error](#)

```
func (e *UnsupportedTypeError) Error() string
```

type UnsupportedValueError

```
type UnsupportedValueError struct {  
    Value reflect.Value  
    Str    string  
}
```

func (*UnsupportedValueError) Error

```
func (e *UnsupportedValueError) Error() string
```

Bugs

This package ignores anonymous (embedded) struct fields during encoding and decoding. A future version may assign meaning to them. To force an anonymous field to be ignored in all future versions of this package, use an explicit ``json:"-"` tag in the struct definition.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package pem

```
import "encoding/pem"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package pem implements the PEM data encoding, which originated in Privacy Enhanced Mail. The most common use of PEM encoding today is in TLS keys and certificates. See RFC 1421.

Index

[func Encode\(out io.Writer, b *Block\) \(err error\)](#)
[func EncodeToMemory\(b *Block\) \[\]byte](#)
[type Block](#)
[func Decode\(data \[\]byte\) \(p *Block, rest \[\]byte\)](#)

Package files

pem.go

func Encode

```
func Encode(out io.Writer, b *Block) (err error)
```

func EncodeToMemory

```
func EncodeToMemory(b *Block) []byte
```

type [Block](#)

```
type Block struct {
    Type      string           // The type, taken from the preamble (
    Headers   map[string]string // Optional headers.
    Bytes     []byte           // The decoded bytes of the contents.
}
```

A Block represents a PEM encoded structure.

The encoded form is:

```
-----BEGIN Type-----
Headers
base64-encoded Bytes
-----END Type-----
```

where Headers is a possibly empty sequence of Key: Value lines.

func [Decode](#)

```
func Decode(data []byte) (p *Block, rest []byte)
```

Decode will find the next PEM formatted block (certificate, private key etc) in the input. It returns that block and the remainder of the input. If no PEM data is found, p is nil and the whole of the input is returned in rest.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package xml

```
import "encoding/xml"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package `xml` implements a simple XML 1.0 parser that understands XML name spaces.

Index

[Constants](#)

[Variables](#)

[func Escape\(w io.Writer, s \[\]byte\)](#)

[func Marshal\(v interface{}\) \(\[\]byte, error\)](#)

[func MarshalIndent\(v interface{}, prefix, indent string\) \(\[\]byte, error\)](#)

[func Unmarshal\(data \[\]byte, v interface{}\) error](#)

[type Attr](#)

[type CharData](#)

[func \(c CharData\) Copy\(\) CharData](#)

[type Comment](#)

[func \(c Comment\) Copy\(\) Comment](#)

[type Decoder](#)

[func NewDecoder\(r io.Reader\) *Decoder](#)

[func \(d *Decoder\) Decode\(v interface{}\) error](#)

[func \(d *Decoder\) DecodeElement\(v interface{}, start *StartElement\)](#)

[error](#)

[func \(d *Decoder\) RawToken\(\) \(Token, error\)](#)

[func \(d *Decoder\) Skip\(\) error](#)

[func \(d *Decoder\) Token\(\) \(t Token, err error\)](#)

[type Directive](#)

[func \(d Directive\) Copy\(\) Directive](#)

[type Encoder](#)

[func NewEncoder\(w io.Writer\) *Encoder](#)

[func \(enc *Encoder\) Encode\(v interface{}\) error](#)

[type EndElement](#)

[type Name](#)

[type ProcInst](#)

[func \(p ProcInst\) Copy\(\) ProcInst](#)

[type StartElement](#)

[func \(e StartElement\) Copy\(\) StartElement](#)

[type SyntaxError](#)

[func \(e *SyntaxError\) Error\(\) string](#)

[type TagPathError](#)

[func \(e *TagPathError\) Error\(\) string](#)

[type Token](#)

[func CopyToken\(t Token\) Token](#)
[type UnmarshalError](#)
[func \(e UnmarshalError\) Error\(\) string](#)
[type UnsupportedTypeError](#)
[func \(e *UnsupportedTypeError\) Error\(\) string](#)
[Bugs](#)

Examples

[MarshalIndent](#)
[Unmarshal](#)

Package files

[marshal.go](#) [read.go](#) [typeinfo.go](#) [xml.go](#)

Constants

```
const (  
    // A generic XML header suitable for use with the output of Mars  
    // This is not automatically added to any output of this package  
    // it is provided as a convenience.  
    Header = `<?xml version="1.0" encoding="UTF-8"?>` + "\n"  
)
```

Variables

```
var HTMLAutoClose = htmlAutoClose
```

HTMLAutoClose is the set of HTML elements that should be considered to close automatically.

```
var HTMLEntity = htmlEntity
```

HTMLEntity is an entity map containing translations for the standard HTML entity characters.

func Escape

```
func Escape(w io.Writer, s []byte)
```

Escape writes to w the properly escaped XML equivalent of the plain text data s.

func [Marshal](#)

```
func Marshal(v interface{}) ([]byte, error)
```

Marshal returns the XML encoding of v.

Marshal handles an array or slice by marshalling each of the elements. Marshal handles a pointer by marshalling the value it points at or, if the pointer is nil, by writing nothing. Marshal handles an interface value by marshalling the value it contains or, if the interface value is nil, by writing nothing. Marshal handles all other data by writing one or more XML elements containing the data.

The name for the XML elements is taken from, in order of preference:

- the tag on the XMLName field, if the data is a struct
- the value of the XMLName field of type xml.Name
- the tag of the struct field used to obtain the data
- the name of the struct field used to obtain the data
- the name of the marshalled type

The XML element for a struct contains marshalled elements for each of the exported fields of the struct, with these exceptions:

- the XMLName field, described above, is omitted.
- a field with tag "-" is omitted.
- a field with tag "name,attr" becomes an attribute with the given name in the XML element.
- a field with tag ",attr" becomes an attribute with the field name in the in the XML element.
- a field with tag ",chardata" is written as character data, not as an XML element.
- a field with tag ",innerxml" is written verbatim, not subject to the usual marshalling procedure.
- a field with tag ",comment" is written as an XML comment, not subject to the usual marshalling procedure. It must not contain the "--" string within it.
- a field with a tag including the "omitempty" option is omitted if the field value is empty. The empty values are false, 0, any nil pointer or interface value, and any array, slice, map, or string of length zero.
- a non-pointer anonymous struct field is handled as if the fields of its value were part of the outer struct.

If a field uses a tag "a>b>c", then the element c will be nested inside parent elements a and b. Fields that appear next to each other that name the same parent will be enclosed in one XML element.

See MarshalIndent for an example.

Marshal will return an error if asked to marshal a channel, function, or map.

func [MarshalIndent](#)

```
func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)
```

MarshalIndent works like Marshal, but each XML element begins on a new indented line that starts with prefix and is followed by one or more copies of indent according to the nesting depth.

? Example

? Example

Code:

```
type Address struct {
    City, State string
}
type Person struct {
    XMLName    xml.Name `xml:"person"`
    Id         int      `xml:"id,attr"`
    FirstName  string   `xml:"name>first"`
    LastName   string   `xml:"name>last"`
    Age        int      `xml:"age"`
    Height     float32  `xml:"height,omitempty"`
    Married    bool
    Address
    Comment string `xml:",comment"`
}

v := &Person{Id: 13, FirstName: "John", LastName: "Doe", Age: 42}
v.Comment = " Need more details. "
v.Address = Address{"Hanga Roa", "Easter Island"}

output, err := xml.MarshalIndent(v, "  ", "    ")
if err != nil {
    fmt.Printf("error: %v\n", err)
}

os.Stdout.Write(output)
```

Output:

```
<person id="13">
    <name>
```

```
    <first>John</first>
    <last>Doe</last>
  </name>
  <age>42</age>
  <Married>>false</Married>
  <City>Hanga Roa</City>
  <State>Easter Island</State>
  <!-- Need more details. -->
</person>
```

func [Unmarshal](#)

```
func Unmarshal(data []byte, v interface{}) error
```

Unmarshal parses the XML-encoded data and stores the result in the value pointed to by `v`, which must be an arbitrary struct, slice, or string. Well-formed data that does not fit into `v` is discarded.

Because Unmarshal uses the reflect package, it can only assign to exported (upper case) fields. Unmarshal uses a case-sensitive comparison to match XML element names to tag values and struct field names.

Unmarshal maps an XML element to a struct using the following rules. In the rules, the tag of a field refers to the value associated with the key 'xml' in the struct field's tag (see the example above).

- * If the struct has a field of type `[]byte` or `string` with tag `",innerxml"`, Unmarshal accumulates the raw XML nested inside the element in that field. The rest of the rules still apply.
- * If the struct has a field named `XMLName` of type `xml.Name`, Unmarshal records the element name in that field.
- * If the `XMLName` field has an associated tag of the form `"name"` or `"namespace-URL name"`, the XML element must have the given name (and, optionally, name space) or else Unmarshal returns an error.
- * If the XML element has an attribute whose name matches a struct field name with an associated tag containing `",attr"` or the explicit name in a struct field tag of the form `"name,attr"`, Unmarshal records the attribute value in that field.
- * If the XML element contains character data, that data is accumulated in the first struct field that has tag `"chardata"`. The struct field may have type `[]byte` or `string`. If there is no such field, the character data is discarded.
- * If the XML element contains comments, they are accumulated in the first struct field that has tag `",comments"`. The struct field may have type `[]byte` or `string`. If there is no such field, the comments are discarded.
- * If the XML element contains a sub-element whose name matches

the prefix of a tag formatted as "a" or "a>b>c", unmarshal will descend into the XML structure looking for elements with the given names, and will map the innermost elements to that struct field. A tag starting with ">" is equivalent to one starting with the field name followed by ">".

- * If the XML element contains a sub-element whose name matches a struct field's XMLName tag and the struct field has no explicit name tag as per the previous rule, unmarshal maps the sub-element to that struct field.
- * If the XML element contains a sub-element whose name matches a field without any mode flags ("attr", "chardata", etc), Unmarsh maps the sub-element to that struct field.
- * If the XML element contains a sub-element that hasn't matched any of the above rules and the struct has a field with tag ",any", unmarshal maps the sub-element to that struct field.
- * A non-pointer anonymous struct field is handled as if the fields of its value were part of the outer struct.
- * A struct field with tag "-" is never unmarshalled into.

Unmarshal maps an XML element to a string or []byte by saving the concatenation of that element's character data in the string or []byte. The saved []byte is never nil.

Unmarshal maps an attribute value to a string or []byte by saving the value in the string or slice.

Unmarshal maps an XML element to a slice by extending the length of the slice and mapping the element to the newly created value.

Unmarshal maps an XML element or attribute value to a bool by setting it to the boolean value represented by the string.

Unmarshal maps an XML element or attribute value to an integer or floating-point field by setting the field to the result of interpreting the string value in decimal. There is no check for overflow.

Unmarshal maps an XML element to an xml.Name by recording the element name.

Unmarshal maps an XML element to a pointer by setting the pointer to a freshly

allocated value and then mapping the element to that value.

? Example

? Example

This example demonstrates unmarshaling an XML excerpt into a value with some preset fields. Note that the Phone field isn't modified and that the XML <Company> element is ignored. Also, the Groups field is assigned considering the element path provided in its tag.

Code:

```
type Email struct {
    Where string `xml:"where,attr"`
    Addr  string
}
type Address struct {
    City, State string
}
type Result struct {
    XMLName xml.Name `xml:"Person"`
    Name    string  `xml:"FullName"`
    Phone   string
    Email   []Email
    Groups  []string `xml:"Group>Value"`
    Address
}
v := Result{Name: "none", Phone: "none"}

data := `
<Person>
  <FullName>Grace R. Emlin</FullName>
  <Company>Example Inc.</Company>
  <Email where="home">
    <Addr>gre@example.com</Addr>
  </Email>
  <Email where='work'>
    <Addr>gre@work.com</Addr>
  </Email>
  <Group>
    <Value>Friends</Value>
    <Value>Squash</Value>
  </Group>
  <City>Hanga Roa</City>
  <State>Easter Island</State>
</Person>
```

```
err := xml.Unmarshal([]byte(data), &v)
if err != nil {
    fmt.Printf("error: %v", err)
    return
}
fmt.Printf("XMLName: %#v\n", v.XMLName)
fmt.Printf("Name: %q\n", v.Name)
fmt.Printf("Phone: %q\n", v.Phone)
fmt.Printf("Email: %v\n", v.Email)
fmt.Printf("Groups: %v\n", v.Groups)
fmt.Printf("Address: %v\n", v.Address)
```

Output:

```
XMLName: xml.Name{Space:"", Local:"Person"}
Name: "Grace R. Emlin"
Phone: "none"
Email: [{home gre@example.com} {work gre@work.com}]
Groups: [Friends Squash]
Address: {Hanga Roa Easter Island}
```

type [Attr](#)

```
type Attr struct {  
    Name Name  
    Value string  
}
```

An Attr represents an attribute in an XML element (Name=Value).

type [CharData](#)

type CharData []byte

A CharData represents XML character data (raw text), in which XML escape sequences have been replaced by the characters they represent.

func (CharData) [Copy](#)

func (c CharData) Copy() CharData

type [Comment](#)

type Comment []byte

A Comment represents an XML comment of the form <!--comment-->. The bytes do not include the <!-- and --> comment markers.

func (Comment) [Copy](#)

func (c Comment) Copy() Comment

type [Decoder](#)

```
type Decoder struct {
    // Strict defaults to true, enforcing the requirements
    // of the XML specification.
    // If set to false, the parser allows input containing common
    // mistakes:
    // * If an element is missing an end tag, the parser invents
    //   end tags as necessary to keep the return values from Token
    //   properly balanced.
    // * In attribute values and character data, unknown or malformed
    //   character entities (sequences beginning with &) are left a
    //
    // Setting:
    //
    // d.Strict = false;
    // d.AutoClose = HTMLAutoClose;
    // d.Entity = HTMLEntity
    //
    // creates a parser that can handle typical HTML.
    Strict bool

    // When Strict == false, AutoClose indicates a set of elements t
    // consider closed immediately after they are opened, regardless
    // of whether an end element is present.
    AutoClose []string

    // Entity can be used to map non-standard entity names to string
    // The parser behaves as if these standard mappings are present
    // regardless of the actual map content:
    //
    // "lt": "<",
    // "gt": ">",
    // "amp": "&",
    // "apos": "'",
    // "quot": `"`
    Entity map[string]string

    // CharsetReader, if non-nil, defines a function to generate
    // charset-conversion readers, converting from the provided
    // non-UTF-8 charset into UTF-8. If CharsetReader is nil or
    // returns an error, parsing stops with an error. One of the
    // the CharsetReader's result values must be non-nil.
    CharsetReader func(charset string, input io.Reader) (io.Reader,
    // contains filtered or unexported fields
}
}
```

A Decoder represents an XML parser reading a particular input stream. The parser assumes that its input is encoded in UTF-8.

func [NewDecoder](#)

```
func NewDecoder(r io.Reader) *Decoder
```

NewDecoder creates a new XML parser reading from r.

func (*Decoder) [Decode](#)

```
func (d *Decoder) Decode(v interface{}) error
```

Decode works like `xml.Unmarshal`, except it reads the decoder stream to find the start element.

func (*Decoder) [DecodeElement](#)

```
func (d *Decoder) DecodeElement(v interface{}, start *StartElement)
```

DecodeElement works like `xml.Unmarshal` except that it takes a pointer to the start XML element to decode into v. It is useful when a client reads some raw XML tokens itself but also wants to defer to `Unmarshal` for some elements.

func (*Decoder) [RawToken](#)

```
func (d *Decoder) RawToken() (Token, error)
```

RawToken is like `Token` but does not verify that start and end elements match and does not translate name space prefixes to their corresponding URLs.

func (*Decoder) [Skip](#)

```
func (d *Decoder) Skip() error
```

Skip reads tokens until it has consumed the end element matching the most recent start element already consumed. It recurs if it encounters a start element, so it can be used to skip nested structures. It returns nil if it finds an end element matching the start element; otherwise it returns an error describing the problem.

func (*Decoder) [Token](#)

```
func (d *Decoder) Token() (t Token, err error)
```

Token returns the next XML token in the input stream. At the end of the input stream, Token returns nil, io.EOF.

Slices of bytes in the returned token data refer to the parser's internal buffer and remain valid only until the next call to Token. To acquire a copy of the bytes, call CopyToken or the token's Copy method.

Token expands self-closing elements such as
 into separate start and end elements returned by successive calls.

Token guarantees that the StartElement and EndElement tokens it returns are properly nested and matched: if Token encounters an unexpected end element, it will return an error.

Token implements XML name spaces as described by <http://www.w3.org/TR/REC-xml-names/>. Each of the Name structures contained in the Token has the Space set to the URL identifying its name space when known. If Token encounters an unrecognized name space prefix, it uses the prefix as the Space rather than report an error.

type [Directive](#)

```
type Directive []byte
```

A Directive represents an XML directive of the form `<!text>`. The bytes do not include the `<!` and `>` markers.

func (Directive) [Copy](#)

```
func (d Directive) Copy() Directive
```

type [Encoder](#)

```
type Encoder struct {  
    // contains filtered or unexported fields  
}
```

An Encoder writes XML data to an output stream.

func [NewEncoder](#)

```
func NewEncoder(w io.Writer) *Encoder
```

NewEncoder returns a new encoder that writes to w.

func (***Encoder**) [Encode](#)

```
func (enc *Encoder) Encode(v interface{}) error
```

Encode writes the XML encoding of v to the stream.

See the documentation for Marshal for details about the conversion of Go values to XML.

type EndElement

```
type EndElement struct {  
    Name Name  
}
```

An EndElement represents an XML end element.

type [Name](#)

```
type Name struct {  
    Space, Local string  
}
```

A Name represents an XML name (Local) annotated with a name space identifier (Space). In tokens returned by `Decoder.Token`, the Space identifier is given as a canonical URL, not the short prefix used in the document being parsed.

type [ProcInst](#)

```
type ProcInst struct {  
    Target string  
    Inst    []byte  
}
```

A ProcInst represents an XML processing instruction of the form <?target inst?>

func (ProcInst) [Copy](#)

```
func (p ProcInst) Copy() ProcInst
```

type [StartElement](#)

```
type StartElement struct {  
    Name Name  
    Attr []Attr  
}
```

A StartElement represents an XML start element.

func (StartElement) [Copy](#)

```
func (e StartElement) Copy() StartElement
```

type [SyntaxError](#)

```
type SyntaxError struct {  
    Msg string  
    Line int  
}
```

A `SyntaxError` represents a syntax error in the XML input stream.

func (*SyntaxError) [Error](#)

```
func (e *SyntaxError) Error() string
```

type [TagPathError](#)

```
type TagPathError struct {  
    Struct      reflect.Type  
    Field1, Tag1 string  
    Field2, Tag2 string  
}
```

A `TagPathError` represents an error in the unmarshalling process caused by the use of field tags with conflicting paths.

func (*TagPathError) [Error](#)

```
func (e *TagPathError) Error() string
```

type [Token](#)

```
type Token interface{}
```

A Token is an interface holding one of the token types: StartElement, EndElement, CharData, Comment, ProcInst, or Directive.

func [CopyToken](#)

```
func CopyToken(t Token) Token
```

CopyToken returns a copy of a Token.

type [UnmarshalError](#)

```
type UnmarshalError string
```

An UnmarshalError represents an error in the unmarshalling process.

func (UnmarshalError) [Error](#)

```
func (e UnmarshalError) Error() string
```

type [UnsupportedTypeError](#)

```
type UnsupportedTypeError struct {  
    Type reflect.Type  
}
```

A MarshalXMLError is returned when Marshal encounters a type that cannot be converted into XML.

func (*UnsupportedTypeError) [Error](#)

```
func (e *UnsupportedTypeError) Error() string
```

Bugs

Mapping between XML elements and data structures is inherently flawed: an XML element is an order-dependent collection of anonymous values, while a data structure is an order-independent collection of named values. See package `json` for a textual representation more suitable to data structures.

Build version `go1.0.1`.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package errors

```
import "errors"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package errors implements functions to manipulate errors.

? Example

? Example

Code:

```
package errors_test

import (
    "fmt"
    "time"
)

// MyError is an error implementation that includes a time and messa
type MyError struct {
    When time.Time
    What string
}

func (e MyError) Error() string {
    return fmt.Sprintf("%v: %v", e.When, e.What)
}

func oops() error {
    return MyError{
        time.Date(1989, 3, 15, 22, 30, 0, 0, time.UTC),
        "the file system has gone away",
    }
}

func Example() {
    if err := oops(); err != nil {
        fmt.Println(err)
    }
    // Output: 1989-03-15 22:30:00 +0000 UTC: the file system has go
}
```

Index

[func New\(text string\) error](#)

Examples

[Package](#)

[New](#)

[New \(Errorf\)](#)

Package files

errors.golang.org

func [New](#)

```
func New(text string) error
```

New returns an error that formats as the given text.

? Example

? Example

Code:

```
err := errors.New("emit macho dwarf: elf header corrupted")
if err != nil {
    fmt.Print(err)
}
```

Output:

```
emit macho dwarf: elf header corrupted
```

? Example (Errorf)

? Example (Errorf)

The `fmt` package's `Errorf` function lets us use the package's formatting features to create descriptive error messages.

Code:

```
const name, id = "bimmler", 17
err := fmt.Errorf("user %q (id %d) not found", name, id)
if err != nil {
    fmt.Print(err)
}
```

Output:

```
user "bimmler" (id 17) not found
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package expvar

```
import "expvar"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `expvar` provides a standardized interface to public variables, such as operation counters in servers. It exposes these variables via HTTP at `/debug/vars` in JSON format.

Operations to set or modify these public variables are atomic.

In addition to adding the HTTP handler, this package registers the following variables:

```
cmdline    os.Args
memstats   runtime.Memstats
```

The package is sometimes only imported for the side effect of registering its HTTP handler and the above variables. To use it this way, link this package into your program:

```
import _ "expvar"
```

Index

[func Do\(f func\(KeyValue\)\)](#)
[func Publish\(name string, v Var\)](#)
[type Float](#)
 [func NewFloat\(name string\) *Float](#)
 [func \(v *Float\) Add\(delta float64\)](#)
 [func \(v *Float\) Set\(value float64\)](#)
 [func \(v *Float\) String\(\) string](#)
[type Func](#)
 [func \(f Func\) String\(\) string](#)
[type Int](#)
 [func NewInt\(name string\) *Int](#)
 [func \(v *Int\) Add\(delta int64\)](#)
 [func \(v *Int\) Set\(value int64\)](#)
 [func \(v *Int\) String\(\) string](#)
[type KeyValue](#)
[type Map](#)
 [func NewMap\(name string\) *Map](#)
 [func \(v *Map\) Add\(key string, delta int64\)](#)
 [func \(v *Map\) AddFloat\(key string, delta float64\)](#)
 [func \(v *Map\) Do\(f func\(KeyValue\)\)](#)
 [func \(v *Map\) Get\(key string\) Var](#)
 [func \(v *Map\) Init\(\) *Map](#)
 [func \(v *Map\) Set\(key string, av Var\)](#)
 [func \(v *Map\) String\(\) string](#)
[type String](#)
 [func NewString\(name string\) *String](#)
 [func \(v *String\) Set\(value string\)](#)
 [func \(v *String\) String\(\) string](#)
[type Var](#)
 [func Get\(name string\) Var](#)

Package files

expvar.go

func [Do](#)

```
func Do(f func(KeyValue))
```

Do calls f for each exported variable. The global variable map is locked during the iteration, but existing entries may be concurrently updated.

func [Publish](#)

```
func Publish(name string, v Var)
```

Publish declares a named exported variable. This should be called from a package's init function when it creates its Vars. If the name is already registered then this will log.Panic.

type [Float](#)

```
type Float struct {  
    // contains filtered or unexported fields  
}
```

Float is a 64-bit float variable that satisfies the Var interface.

func [NewFloat](#)

```
func NewFloat(name string) *Float
```

func (*Float) [Add](#)

```
func (v *Float) Add(delta float64)
```

Add adds delta to v.

func (*Float) [Set](#)

```
func (v *Float) Set(value float64)
```

Set sets v to value.

func (*Float) [String](#)

```
func (v *Float) String() string
```

type [Func](#)

```
type Func func() interface{}
```

Func implements Var by calling the function and formatting the returned value using JSON.

func (Func) [String](#)

```
func (f Func) String() string
```

type [Int](#)

```
type Int struct {  
    // contains filtered or unexported fields  
}
```

Int is a 64-bit integer variable that satisfies the Var interface.

func [NewInt](#)

```
func NewInt(name string) *Int
```

func ([*Int](#)) [Add](#)

```
func (v *Int) Add(delta int64)
```

func ([*Int](#)) [Set](#)

```
func (v *Int) Set(value int64)
```

func ([*Int](#)) [String](#)

```
func (v *Int) String() string
```

type [KeyValue](#)

```
type KeyValue struct {  
    Key    string  
    Value  Var  
}
```

KeyValue represents a single entry in a Map.

type [Map](#)

```
type Map struct {  
    // contains filtered or unexported fields  
}
```

Map is a string-to-Var map variable that satisfies the Var interface.

func [NewMap](#)

```
func NewMap(name string) *Map
```

func (***Map**) [Add](#)

```
func (v *Map) Add(key string, delta int64)
```

func (***Map**) [AddFloat](#)

```
func (v *Map) AddFloat(key string, delta float64)
```

AddFloat adds delta to the *Float value stored under the given map key.

func (***Map**) [Do](#)

```
func (v *Map) Do(f func(KeyValue))
```

Do calls f for each entry in the map. The map is locked during the iteration, but existing entries may be concurrently updated.

func (***Map**) [Get](#)

```
func (v *Map) Get(key string) Var
```

func (***Map**) [Init](#)

```
func (v *Map) Init() *Map
```

func (***Map**) [Set](#)

```
func (v *Map) Set(key string, av Var)
```

```
func (*Map) String
```

```
func (v *Map) String() string
```

type [String](#)

```
type String struct {  
    // contains filtered or unexported fields  
}
```

String is a string variable, and satisfies the Var interface.

func [NewString](#)

```
func NewString(name string) *String
```

func (*String) [Set](#)

```
func (v *String) Set(value string)
```

func (*String) [String](#)

```
func (v *String) String() string
```

type [Var](#)

```
type Var interface {  
    String() string  
}
```

Var is an abstract type for all exported variables.

func [Get](#)

```
func Get(name string) Var
```

Get retrieves a named exported variable.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package flag

```
import "flag"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package flag implements command-line flag parsing.

Usage:

Define flags using flag.String(), Bool(), Int(), etc.

This declares an integer flag, -flagname, stored in the pointer ip, with type *int.

```
import "flag"
var ip = flag.Int("flagname", 1234, "help message for flagname")
```

If you like, you can bind the flag to a variable using the Var() functions.

```
var flagvar int
func init() {
    flag.IntVar(&flagvar, "flagname", 1234, "help message for fl
}
```

Or you can create custom flags that satisfy the Value interface (with pointer receivers) and couple them to flag parsing by

```
flag.Var(&flagVal, "name", "help message for flagname")
```

For such flags, the default value is just the initial value of the variable.

After all flags are defined, call

```
flag.Parse()
```

to parse the command line into the defined flags.

Flags may then be used directly. If you're using the flags themselves, they are all pointers; if you bind to variables, they're values.

```
fmt.Println("ip has value ", *ip)
fmt.Println("flagvar has value ", flagvar)
```

After parsing, the arguments after the flag are available as the slice flag.Args() or individually as flag.Arg(i). The arguments are indexed from 0 up to flag.NArg().

Command line flag syntax:

```
-flag  
-flag=x  
-flag x // non-boolean flags only
```

One or two minus signs may be used; they are equivalent. The last form is not permitted for boolean flags because the meaning of the command

```
cmd -x *
```

will change if there is a file called 0, false, etc. You must use the `-flag=false` form to turn off a boolean flag.

Flag parsing stops just before the first non-flag argument ("`-`" is a non-flag argument) or after the terminator "`--`".

Integer flags accept 1234, 0664, 0x1234 and may be negative. Boolean flags may be 1, 0, t, f, true, false, TRUE, FALSE, True, False. Duration flags accept any input valid for `time.ParseDuration`.

The default set of command-line flags is controlled by top-level functions. The `FlagSet` type allows one to define independent sets of flags, such as to implement subcommands in a command-line interface. The methods of `FlagSet` are analogous to the top-level functions for the command-line flag set.

? Example

? Example

Code:

```
// These examples demonstrate more intricate uses of the flag package  
package flag_test
```

```
import (  
    "errors"  
    "flag"  
    "fmt"  
    "strings"  
    "time"  
)
```

```
// Example 1: A single string flag called "species" with default val
```

```

var species = flag.String("species", "gopher", "the species we are s

// Example 2: Two flags sharing a variable, so we can have a shortha
// The order of initialization is undefined, so make sure both use t
// same default value. They must be set up with an init function.
var gopherType string

func init() {
    const (
        defaultGopher = "pocket"
        usage          = "the variety of gopher"
    )
    flag.StringVar(&gopherType, "gopher_type", defaultGopher, usage)
    flag.StringVar(&gopherType, "g", defaultGopher, usage+" (shortha
}

// Example 3: A user-defined flag type, a slice of durations.
type interval []time.Duration

// String is the method to format the flag's value, part of the flag
// The String method's output will be used in diagnostics.
func (i *interval) String() string {
    return fmt.Sprint(*i)
}

// Set is the method to set the flag value, part of the flag.Value i
// Set's argument is a string to be parsed to set the flag.
// It's a comma-separated list, so we split it.
func (i *interval) Set(value string) error {
    // If we wanted to allow the flag to be set multiple times,
    // accumulating values, we would delete this if statement.
    // That would permit usages such as
    // -deltaT 10s -deltaT 15s
    // and other combinations.
    if len(*i) > 0 {
        return errors.New("interval flag already set")
    }
    for _, dt := range strings.Split(value, ",") {
        duration, err := time.ParseDuration(dt)
        if err != nil {
            return err
        }
        *i = append(*i, duration)
    }
    return nil
}

// Define a flag to accumulate durations. Because it has a special t
// we need to use the Var function and therefore create the flag dur
// init.

```

```
var intervalFlag interval

func init() {
    // Tie the command-line flag to the intervalFlag variable and
    // set a usage message.
    flag.Var(&intervalFlag, "deltaT", "comma-separated list of inter
}

func Example() {
    // All the interesting pieces are with the variables declared ab
    // to enable the flag package to see the flags defined there, on
    // execute, typically at the start of main (not init!):
    // flag.Parse()
    // We don't run it here because this is not a main function and
    // the testing suite has already parsed the flags.
}
```

Index

Variables

func Arg(i int) string

func Args() []string

func Bool(name string, value bool, usage string) *bool

func BoolVar(p *bool, name string, value bool, usage string)

func Duration(name string, value time.Duration, usage string)

*time.Duration

func DurationVar(p *time.Duration, name string, value time.Duration, usage string)

func Float64(name string, value float64, usage string) *float64

func Float64Var(p *float64, name string, value float64, usage string)

func Int(name string, value int, usage string) *int

func Int64(name string, value int64, usage string) *int64

func Int64Var(p *int64, name string, value int64, usage string)

func IntVar(p *int, name string, value int, usage string)

func NArg() int

func NFlag() int

func Parse()

func Parsed() bool

func PrintDefaults()

func Set(name, value string) error

func String(name string, value string, usage string) *string

func StringVar(p *string, name string, value string, usage string)

func Uint(name string, value uint, usage string) *uint

func Uint64(name string, value uint64, usage string) *uint64

func Uint64Var(p *uint64, name string, value uint64, usage string)

func UintVar(p *uint, name string, value uint, usage string)

func Var(value Value, name string, usage string)

func Visit(fn func(*Flag))

func VisitAll(fn func(*Flag))

type ErrorHandling

type Flag

func Lookup(name string) *Flag

type FlagSet

func NewFlagSet(name string, errorHandling ErrorHandling) *FlagSet

func (f *FlagSet) Arg(i int) string
func (f *FlagSet) Args() []string
func (f *FlagSet) Bool(name string, value bool, usage string) *bool
func (f *FlagSet) BoolVar(p *bool, name string, value bool, usage string)
func (f *FlagSet) Duration(name string, value time.Duration, usage string) *time.Duration
func (f *FlagSet) DurationVar(p *time.Duration, name string, value time.Duration, usage string)
func (f *FlagSet) Float64(name string, value float64, usage string) *float64
func (f *FlagSet) Float64Var(p *float64, name string, value float64, usage string)
func (f *FlagSet) Init(name string, errorHandler ErrorHandling)
func (f *FlagSet) Int(name string, value int, usage string) *int
func (f *FlagSet) Int64(name string, value int64, usage string) *int64
func (f *FlagSet) Int64Var(p *int64, name string, value int64, usage string)
func (f *FlagSet) IntVar(p *int, name string, value int, usage string)
func (f *FlagSet) Lookup(name string) *Flag
func (f *FlagSet) NArg() int
func (f *FlagSet) NFlag() int
func (f *FlagSet) Parse(arguments []string) error
func (f *FlagSet) Parsed() bool
func (f *FlagSet) PrintDefaults()
func (f *FlagSet) Set(name, value string) error
func (f *FlagSet) SetOutput(output io.Writer)
func (f *FlagSet) String(name string, value string, usage string) *string
func (f *FlagSet) StringVar(p *string, name string, value string, usage string)
func (f *FlagSet) Uint(name string, value uint, usage string) *uint
func (f *FlagSet) Uint64(name string, value uint64, usage string) *uint64
func (f *FlagSet) Uint64Var(p *uint64, name string, value uint64, usage string)
func (f *FlagSet) UintVar(p *uint, name string, value uint, usage string)
func (f *FlagSet) Var(value Value, name string, usage string)
func (f *FlagSet) Visit(fn func(*Flag))
func (f *FlagSet) VisitAll(fn func(*Flag))
type Value

Examples

[Package](#)

Package files

[flag.go](#)

Variables

```
var ErrHelp = errors.New("flag: help requested")
```

ErrHelp is the error returned if the flag -help is invoked but no such flag is defined.

```
var Usage = func() {  
    fmt.Fprintf(os.Stderr, "Usage of %s:\n", os.Args[0])  
    PrintDefaults()  
}
```

Usage prints to standard error a usage message documenting all defined command-line flags. The function is a variable that may be changed to point to a custom function.

func [Arg](#)

```
func Arg(i int) string
```

Arg returns the i'th command-line argument. Arg(0) is the first remaining argument after flags have been processed.

func [Args](#)

```
func Args() []string
```

Args returns the non-flag command-line arguments.

func Bool

```
func Bool(name string, value bool, usage string) *bool
```

Bool defines a bool flag with specified name, default value, and usage string.
The return value is the address of a bool variable that stores the value of the flag.

func BoolVar

```
func BoolVar(p *bool, name string, value bool, usage string)
```

BoolVar defines a bool flag with specified name, default value, and usage string. The argument p points to a bool variable in which to store the value of the flag.

func Duration

```
func Duration(name string, value time.Duration, usage string) *time.
```

Duration defines a time.Duration flag with specified name, default value, and usage string. The return value is the address of a time.Duration variable that stores the value of the flag.

func DurationVar

```
func DurationVar(p *time.Duration, name string, value time.Duration,
```

DurationVar defines a time.Duration flag with specified name, default value, and usage string. The argument p points to a time.Duration variable in which to store the value of the flag.

func [Float64](#)

```
func Float64(name string, value float64, usage string) *float64
```

Float64 defines a float64 flag with specified name, default value, and usage string. The return value is the address of a float64 variable that stores the value of the flag.

func Float64Var

```
func Float64Var(p *float64, name string, value float64, usage string
```

Float64Var defines a float64 flag with specified name, default value, and usage string. The argument p points to a float64 variable in which to store the value of the flag.

func [Int](#)

```
func Int(name string, value int, usage string) *int
```

Int defines an int flag with specified name, default value, and usage string. The return value is the address of an int variable that stores the value of the flag.

func [Int64](#)

```
func Int64(name string, value int64, usage string) *int64
```

Int64 defines an int64 flag with specified name, default value, and usage string. The return value is the address of an int64 variable that stores the value of the flag.

func [Int64Var](#)

```
func Int64Var(p *int64, name string, value int64, usage string)
```

Int64Var defines an int64 flag with specified name, default value, and usage string. The argument p points to an int64 variable in which to store the value of the flag.

func [IntVar](#)

```
func IntVar(p *int, name string, value int, usage string)
```

IntVar defines an int flag with specified name, default value, and usage string. The argument p points to an int variable in which to store the value of the flag.

func NArg

```
func NArg() int
```

NArg is the number of arguments remaining after flags have been processed.

func [NFlag](#)

```
func NFlag() int
```

NFlag returns the number of command-line flags that have been set.

func [Parse](#)

`func Parse()`

Parse parses the command-line flags from `os.Args[1:]`. Must be called after all flags are defined and before flags are accessed by the program.

func Parsed

```
func Parsed() bool
```

Parsed returns true if the command-line flags have been parsed.

func PrintDefaults

```
func PrintDefaults()
```

PrintDefaults prints to standard error the default values of all defined command-line flags.

func [Set](#)

```
func Set(name, value string) error
```

Set sets the value of the named command-line flag.

func [String](#)

```
func String(name string, value string, usage string) *string
```

String defines a string flag with specified name, default value, and usage string. The return value is the address of a string variable that stores the value of the flag.

func StringVar

```
func StringVar(p *string, name string, value string, usage string)
```

StringVar defines a string flag with specified name, default value, and usage string. The argument p points to a string variable in which to store the value of the flag.

func [Uint](#)

```
func Uint(name string, value uint, usage string) *uint
```

Uin defines a uint flag with specified name, default value, and usage string. The return value is the address of a uint variable that stores the value of the flag.

func [Uint64](#)

```
func Uint64(name string, value uint64, usage string) *uint64
```

Uin64 defines a uint64 flag with specified name, default value, and usage string. The return value is the address of a uint64 variable that stores the value of the flag.

func [Uin64Var](#)

```
func Uin64Var(p *uint64, name string, value uint64, usage string)
```

Uin64Var defines a uint64 flag with specified name, default value, and usage string. The argument p points to a uint64 variable in which to store the value of the flag.

func UinVar

```
func UinVar(p *uint, name string, value uint, usage string)
```

UinVar defines a uint flag with specified name, default value, and usage string. The argument p points to a uint variable in which to store the value of the flag.

func [Var](#)

```
func Var(value Value, name string, usage string)
```

Var defines a flag with the specified name and usage string. The type and value of the flag are represented by the first argument, of type Value, which typically holds a user-defined implementation of Value. For instance, the caller could create a flag that turns a comma-separated string into a slice of strings by giving the slice the methods of Value; in particular, Set would decompose the comma-separated string into the slice.

func [Visit](#)

```
func Visit(fn func(*Flag))
```

Visit visits the command-line flags in lexicographical order, calling fn for each. It visits only those flags that have been set.

func [VisitAll](#)

```
func VisitAll(fn func(*Flag))
```

VisitAll visits the command-line flags in lexicographical order, calling fn for each. It visits all flags, even those not set.

type [ErrorHandling](#)

```
type ErrorHandling int
```

ErrorHandling defines how to handle flag parsing errors.

```
const (  
    ContinueOnError ErrorHandling = iota  
    ExitOnError  
    PanicOnError  
)
```

type [Flag](#)

```
type Flag struct {
    Name      string // name as it appears on command line
    Usage     string // help message
    Value     Value  // value as set
    DefValue  string // default value (as text); for usage message
}
```

A Flag represents the state of a flag.

func [Lookup](#)

```
func Lookup(name string) *Flag
```

Lookup returns the Flag structure of the named command-line flag, returning nil if none exists.

type [FlagSet](#)

```
type FlagSet struct {  
    // Usage is the function called when an error occurs while parsing  
    // The field is a function (not a method) that may be changed to  
    // a custom error handler.  
    Usage func()  
    // contains filtered or unexported fields  
}
```

A FlagSet represents a set of defined flags.

func [NewFlagSet](#)

```
func NewFlagSet(name string, errorHandler ErrorHandling) *FlagSet
```

NewFlagSet returns a new, empty flag set with the specified name and error handling property.

func (*FlagSet) [Arg](#)

```
func (f *FlagSet) Arg(i int) string
```

Arg returns the i'th argument. Arg(0) is the first remaining argument after flags have been processed.

func (*FlagSet) [Args](#)

```
func (f *FlagSet) Args() []string
```

Args returns the non-flag arguments.

func (*FlagSet) [Bool](#)

```
func (f *FlagSet) Bool(name string, value bool, usage string) *bool
```

Bool defines a bool flag with specified name, default value, and usage string. The return value is the address of a bool variable that stores the value of the flag.

func (*FlagSet) [BoolVar](#)

```
func (f *FlagSet) BoolVar(p *bool, name string, value bool, usage st
```

BoolVar defines a bool flag with specified name, default value, and usage string. The argument p points to a bool variable in which to store the value of the flag.

func (*FlagSet) [Duration](#)

```
func (f *FlagSet) Duration(name string, value time.Duration, usage s
```

Duration defines a time.Duration flag with specified name, default value, and usage string. The return value is the address of a time.Duration variable that stores the value of the flag.

func (*FlagSet) [DurationVar](#)

```
func (f *FlagSet) DurationVar(p *time.Duration, name string, value t
```

DurationVar defines a time.Duration flag with specified name, default value, and usage string. The argument p points to a time.Duration variable in which to store the value of the flag.

func (*FlagSet) [Float64](#)

```
func (f *FlagSet) Float64(name string, value float64, usage string)
```

Float64 defines a float64 flag with specified name, default value, and usage string. The return value is the address of a float64 variable that stores the value of the flag.

func (*FlagSet) [Float64Var](#)

```
func (f *FlagSet) Float64Var(p *float64, name string, value float64,
```

Float64Var defines a float64 flag with specified name, default value, and usage string. The argument p points to a float64 variable in which to store the value of the flag.

func (*FlagSet) [Init](#)

```
func (f *FlagSet) Init(name string, errorHandler ErrorHandling)
```

Init sets the name and error handling property for a flag set. By default, the zero FlagSet uses an empty name and the ContinueOnError error handling policy.

func (*FlagSet) [Int](#)

```
func (f *FlagSet) Int(name string, value int, usage string) *int
```

Int defines an int flag with specified name, default value, and usage string. The return value is the address of an int variable that stores the value of the flag.

func (*FlagSet) [Int64](#)

```
func (f *FlagSet) Int64(name string, value int64, usage string) *int
```

Int64 defines an int64 flag with specified name, default value, and usage string. The return value is the address of an int64 variable that stores the value of the flag.

func (*FlagSet) [Int64Var](#)

```
func (f *FlagSet) Int64Var(p *int64, name string, value int64, usage
```

Int64Var defines an int64 flag with specified name, default value, and usage string. The argument p points to an int64 variable in which to store the value of the flag.

func (*FlagSet) [IntVar](#)

```
func (f *FlagSet) IntVar(p *int, name string, value int, usage strin
```

IntVar defines an int flag with specified name, default value, and usage string. The argument p points to an int variable in which to store the value of the flag.

func (*FlagSet) [Lookup](#)

```
func (f *FlagSet) Lookup(name string) *Flag
```

Lookup returns the Flag structure of the named flag, returning nil if none exists.

func (*FlagSet) [NArg](#)

```
func (f *FlagSet) NArg() int
```

NArg is the number of arguments remaining after flags have been processed.

func (*FlagSet) [NFlag](#)

```
func (f *FlagSet) NFlag() int
```

NFlag returns the number of flags that have been set.

func (*FlagSet) [Parse](#)

```
func (f *FlagSet) Parse(arguments []string) error
```

Parse parses flag definitions from the argument list, which should not include the command name. Must be called after all flags in the FlagSet are defined and before flags are accessed by the program. The return value will be ErrHelp if -help was set but not defined.

func (*FlagSet) [Parsed](#)

```
func (f *FlagSet) Parsed() bool
```

Parsed reports whether f.Parse has been called.

func (*FlagSet) [PrintDefaults](#)

```
func (f *FlagSet) PrintDefaults()
```

PrintDefaults prints, to standard error unless configured otherwise, the default values of all defined flags in the set.

func (*FlagSet) [Set](#)

```
func (f *FlagSet) Set(name, value string) error
```

Set sets the value of the named flag.

func (*FlagSet) [SetOutput](#)

```
func (f *FlagSet) SetOutput(output io.Writer)
```

SetOutput sets the destination for usage and error messages. If output is nil, os.Stderr is used.

func (*FlagSet) [String](#)

```
func (f *FlagSet) String(name string, value string, usage string) *s
```

String defines a string flag with specified name, default value, and usage string. The return value is the address of a string variable that stores the value of the flag.

func (*FlagSet) [StringVar](#)

```
func (f *FlagSet) StringVar(p *string, name string, value string, us
```

StringVar defines a string flag with specified name, default value, and usage string. The argument p points to a string variable in which to store the value of the flag.

func (*FlagSet) [Uint](#)

```
func (f *FlagSet) Uint(name string, value uint, usage string) *uint
```

Uint defines a uint flag with specified name, default value, and usage string. The return value is the address of a uint variable that stores the value of the flag.

func (*FlagSet) [Uint64](#)

```
func (f *FlagSet) Uint64(name string, value uint64, usage string) *u
```

Uint64 defines a uint64 flag with specified name, default value, and usage string. The return value is the address of a uint64 variable that stores the value of the flag.

func (*FlagSet) [Uint64Var](#)

```
func (f *FlagSet) Uint64Var(p *uint64, name string, value uint64, us
```

Uint64Var defines a uint64 flag with specified name, default value, and usage string. The argument p points to a uint64 variable in which to store the value of the flag.

func (*FlagSet) [UintVar](#)

```
func (f *FlagSet) UintVar(p *uint, name string, value uint, usage st
```

UintVar defines a uint flag with specified name, default value, and usage string. The argument p points to a uint variable in which to store the value of the flag.

func (*FlagSet) [Var](#)

```
func (f *FlagSet) Var(value Value, name string, usage string)
```

Var defines a flag with the specified name and usage string. The type and value of the flag are represented by the first argument, of type Value, which typically holds a user-defined implementation of Value. For instance, the caller could create a flag that turns a comma-separated string into a slice of strings by giving the slice the methods of Value; in particular, Set would decompose the comma-separated string into the slice.

func (*FlagSet) [Visit](#)

```
func (f *FlagSet) Visit(fn func(*Flag))
```

Visit visits the flags in lexicographical order, calling fn for each. It visits only those flags that have been set.

func (*FlagSet) [VisitAll](#)

```
func (f *FlagSet) VisitAll(fn func(*Flag))
```

VisitAll visits the flags in lexicographical order, calling fn for each. It visits all flags, even those not set.

type [Value](#)

```
type Value interface {  
    String() string  
    Set(string) error  
}
```

Value is the interface to the dynamic value stored in a flag. (The default value is represented as a string.)

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package fmt

```
import "fmt"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `fmt` implements formatted I/O with functions analogous to C's `printf` and `scanf`. The format 'verbs' are derived from C's but are simpler.

Printing

The verbs:

General:

```
%v      the value in a default format.
         when printing structs, the plus flag (%+v) adds field names
%#v     a Go-syntax representation of the value
%T      a Go-syntax representation of the type of the value
%%      a literal percent sign; consumes no value
```

Boolean:

```
%t      the word true or false
```

Integer:

```
%b      base 2
%c      the character represented by the corresponding Unicode code
%d      base 10
%o      base 8
%q      a single-quoted character literal safely escaped with Go syn
%x      base 16, with lower-case letters for a-f
%X      base 16, with upper-case letters for A-F
%U      Unicode format: U+1234; same as "U+%04X"
```

Floating-point and complex constituents:

```
%b      decimalless scientific notation with exponent a power of two
         in the manner of strconv.FormatFloat with the 'b' format,
         e.g. -123456p-78
%e      scientific notation, e.g. -1234.456e+78
%E      scientific notation, e.g. -1234.456E+78
%f      decimal point but no exponent, e.g. 123.456
%g      whichever of %e or %f produces more compact output
%G      whichever of %E or %f produces more compact output
```

String and slice of bytes:

<code>%s</code>	the uninterpreted bytes of the string or slice
<code>%q</code>	a double-quoted string safely escaped with Go syntax
<code>%x</code>	base 16, lower-case, two characters per byte
<code>%X</code>	base 16, upper-case, two characters per byte

Pointer:

<code>%p</code>	base 16 notation, with leading <code>0x</code>
-----------------	--

There is no 'u' flag. Integers are printed unsigned if they have unsigned type. Similarly, there is no need to specify the size of the operand (`int8`, `int64`).

The width and precision control formatting and are in units of Unicode code points. (This differs from C's `printf` where the units are numbers of bytes.) Either or both of the flags may be replaced with the character '*', causing their values to be obtained from the next operand, which must be of type `int`.

For numeric values, width sets the width of the field and precision sets the number of places after the decimal, if appropriate. For example, the format `%6.2f` prints 123.45.

For strings, width is the minimum number of characters to output, padding with spaces if necessary, and precision is the maximum number of characters to output, truncating if necessary.

Other flags:

<code>+</code>	always print a sign for numeric values; guarantee ASCII-only output for <code>%q</code> (<code> %+q</code>)
<code>-</code>	pad with spaces on the right rather than the left (left-just)
<code>#</code>	alternate format: add leading <code>0</code> for octal (<code> %#o</code>), <code>0x</code> for hex <code>0X</code> for hex (<code> %#X</code>); suppress <code>0x</code> for <code>%p</code> (<code> %#p</code>); print a raw (backquoted) string if possible for <code>%q</code> (<code> %#q</code>); write e.g. <code>U+0078 'x'</code> if the character is printable for <code>%U</code> (
<code>' '</code>	(space) leave a space for elided sign in numbers (<code> % d</code>); put spaces between bytes printing strings or slices in hex (
<code>0</code>	pad with leading zeros rather than spaces

For each `Printf`-like function, there is also a `Print` function that takes no format and is equivalent to saying `%v` for every operand. Another variant `Println` inserts blanks between operands and appends a newline.

Regardless of the verb, if an operand is an interface value, the internal concrete value is used, not the interface itself. Thus:

```
var i interface{} = 23
fmt.Printf("%v\n", i)
```

will print 23.

If an operand implements interface `Formatter`, that interface can be used for fine control of formatting.

If the format (which is implicitly `%v` for `Println` etc.) is valid for a string (`%s %q %v %x %X`), the following two rules also apply:

1. If an operand implements the error interface, the `Error` method will be used to convert the object to a string, which will then be formatted as required by the verb (if any).
2. If an operand implements method `String()` string, that method will be used to convert the object to a string, which will then be formatted as required by the verb (if any).

To avoid recursion in cases such as

```
type X string
func (x X) String() string { return Sprintf("<%s>", x) }
```

convert the value before recurring:

```
func (x X) String() string { return Sprintf("<%s>", string(x)) }
```

Format errors:

If an invalid argument is given for a verb, such as providing a string to `%d`, the generated string will contain a description of the problem, as in these examples:

```
Wrong type or unknown verb: %!verb(type=value)
    Printf("%d", hi):           %!d(string=hi)
Too many arguments: %!(EXTRA type=value)
    Printf("hi", "guys"):      hi%!(EXTRA string=guys)
Too few arguments: %!verb(MISSING)
    Printf("hi%d"):           hi %!d(MISSING)
Non-int for width or precision: %!(BADWIDTH) or %!(BADPREC)
```

```
Printf("%*s", 4.5, "hi"): %(BADWIDTH)hi  
Printf("%. *s", 4.5, "hi"): %(BADPREC)hi
```

All errors begin with the string "%!" followed sometimes by a single character (the verb) and end with a parenthesized description.

Scanning

An analogous set of functions scans formatted text to yield values. Scan, Scanf and Scanln read from os.Stdin; Fscan, Fscanf and Fscanln read from a specified io.Reader; Sscan, Sscanf and Sscanln read from an argument string. Scanln, Fscanln and Sscanln stop scanning at a newline and require that the items be followed by one; Sscanf, Fscanf and Sscanf require newlines in the input to match newlines in the format; the other routines treat newlines as spaces.

Scanf, Fscanf, and Sscanf parse the arguments according to a format string, analogous to that of Printf. For example, %x will scan an integer as a hexadecimal number, and %v will scan the default representation format for the value.

The formats behave analogously to those of Printf with the following exceptions:

```
%p is not implemented  
%T is not implemented  
%e %E %f %F %g %G are all equivalent and scan any floating point or  
%s and %v on strings scan a space-delimited token
```

The familiar base-setting prefixes 0 (octal) and 0x (hexadecimal) are accepted when scanning integers without a format or with the %v verb.

Width is interpreted in the input text (%5s means at most five runes of input will be read to scan a string) but there is no syntax for scanning with a precision (no %5.2f, just %5f).

When scanning with a format, all non-empty runs of space characters (except newline) are equivalent to a single space in both the format and the input. With that proviso, text in the format string must match the input text; scanning stops if it does not, with the return value of the function indicating the number of arguments scanned.

In all the scanning functions, if an operand implements method Scan (that is, it

implements the Scanner interface) that method will be used to scan the text for that operand. Also, if the number of arguments scanned is less than the number of arguments provided, an error is returned.

All arguments to be scanned must be either pointers to basic types or implementations of the Scanner interface.

Note: Fscan etc. can read one character (rune) past the input they return, which means that a loop calling a scan routine may skip some of the input. This is usually a problem only when there is no space between input values. If the reader provided to Fscan implements ReadRune, that method will be used to read characters. If the reader also implements UnreadRune, that method will be used to save the character and successive calls will not lose data. To attach ReadRune and UnreadRune methods to a reader without that capability, use bufio.NewReader.

Index

[func Errorf\(format string, a ...interface{}\) error](#)
[func Fprint\(w io.Writer, a ...interface{}\) \(n int, err error\)](#)
[func Fprintf\(w io.Writer, format string, a ...interface{}\) \(n int, err error\)](#)
[func Fprintln\(w io.Writer, a ...interface{}\) \(n int, err error\)](#)
[func Fscan\(r io.Reader, a ...interface{}\) \(n int, err error\)](#)
[func Fscanf\(r io.Reader, format string, a ...interface{}\) \(n int, err error\)](#)
[func Fscanln\(r io.Reader, a ...interface{}\) \(n int, err error\)](#)
[func Print\(a ...interface{}\) \(n int, err error\)](#)
[func Printf\(format string, a ...interface{}\) \(n int, err error\)](#)
[func Println\(a ...interface{}\) \(n int, err error\)](#)
[func Scan\(a ...interface{}\) \(n int, err error\)](#)
[func Scanf\(format string, a ...interface{}\) \(n int, err error\)](#)
[func Scanln\(a ...interface{}\) \(n int, err error\)](#)
[func Sprint\(a ...interface{}\) string](#)
[func Sprintf\(format string, a ...interface{}\) string](#)
[func Sprintln\(a ...interface{}\) string](#)
[func Sscan\(str string, a ...interface{}\) \(n int, err error\)](#)
[func Sscanf\(str string, format string, a ...interface{}\) \(n int, err error\)](#)
[func Sscanln\(str string, a ...interface{}\) \(n int, err error\)](#)
[type Formatter](#)
[type GoStringer](#)
[type ScanState](#)
[type Scanner](#)
[type State](#)
[type Stringer](#)

Package files

[doc.go](#) [format.go](#) [print.go](#) [scan.go](#)

func [Errorf](#)

```
func Errorf(format string, a ...interface{}) error
```

Errorf formats according to a format specifier and returns the string as a value that satisfies error.

func [Fprint](#)

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
```

Fprint formats using the default formats for its operands and writes to w. Spaces are added between operands when neither is a string. It returns the number of bytes written and any write error encountered.

func [Fprintf](#)

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, e
```

Fprintf formats according to a format specifier and writes to w. It returns the number of bytes written and any write error encountered.

func [Fprintln](#)

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
```

Fprintln formats using the default formats for its operands and writes to w. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.

func [Fscan](#)

```
func Fscan(r io.Reader, a ...interface{}) (n int, err error)
```

Fscan scans text read from `r`, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, `err` will report why.

func [Fscanf](#)

```
func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)
```

Fscanf scans text read from r, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully parsed.

func [Fscanln](#)

```
func Fscanln(r io.Reader, a ...interface{}) (n int, err error)
```

Fscanln is similar to Fscan, but stops scanning at a newline and after the final item there must be a newline or EOF.

func [Print](#)

```
func Print(a ...interface{}) (n int, err error)
```

Print formats using the default formats for its operands and writes to standard output. Spaces are added between operands when neither is a string. It returns the number of bytes written and any write error encountered.

func [Printf](#)

```
func Printf(format string, a ...interface{}) (n int, err error)
```

Printf formats according to a format specifier and writes to standard output. It returns the number of bytes written and any write error encountered.

func [Println](#)

```
func Println(a ...interface{}) (n int, err error)
```

Println formats using the default formats for its operands and writes to standard output. Spaces are always added between operands and a newline is appended. It returns the number of bytes written and any write error encountered.

func Scan

```
func Scan(a ...interface{}) (n int, err error)
```

Scan scans text read from standard input, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, err will report why.

func [Scanf](#)

```
func Scanf(format string, a ...interface{}) (n int, err error)
```

Scanf scans text read from standard input, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully scanned.

func [Scanln](#)

```
func Scanln(a ...interface{}) (n int, err error)
```

Scanln is similar to Scan, but stops scanning at a newline and after the final item there must be a newline or EOF.

func [Sprintf](#)

```
func Sprintf(a ...interface{}) string
```

Sprintf formats using the default formats for its operands and returns the resulting string. Spaces are added between operands when neither is a string.

func [Sprintf](#)

```
func Sprintf(format string, a ...interface{}) string
```

Sprintf formats according to a format specifier and returns the resulting string.

func Sprintln

```
func Sprintln(a ...interface{}) string
```

Sprintln formats using the default formats for its operands and returns the resulting string. Spaces are always added between operands and a newline is appended.

func [Sscan](#)

```
func Sscan(str string, a ...interface{}) (n int, err error)
```

Sscan scans the argument string, storing successive space-separated values into successive arguments. Newlines count as space. It returns the number of items successfully scanned. If that is less than the number of arguments, err will report why.

func [Sscanf](#)

```
func Sscanf(str string, format string, a ...interface{}) (n int, err
```

Sscanf scans the argument string, storing successive space-separated values into successive arguments as determined by the format. It returns the number of items successfully parsed.

func [Sscanln](#)

```
func Sscanln(str string, a ...interface{}) (n int, err error)
```

Sscanln is similar to Sscan, but stops scanning at a newline and after the final item there must be a newline or EOF.

type [Formatter](#)

```
type Formatter interface {  
    Format(f State, c rune)  
}
```

Formatter is the interface implemented by values with a custom formatter. The implementation of Format may call Sprintf or Fprintf(f) etc. to generate its output.

type [GoStringer](#)

```
type GoStringer interface {  
    GoString() string  
}
```

GoStringer is implemented by any value that has a GoString method, which defines the Go syntax for that value. The GoString method is used to print values passed as an operand to a %#v format.

type [ScanState](#)

```
type ScanState interface {
    // ReadRune reads the next rune (Unicode code point) from the in
    // If invoked during Scanln, Fscanln, or Sscanln, ReadRune() wil
    // return EOF after returning the first '\n' or when reading bey
    // the specified width.
    ReadRune() (r rune, size int, err error)
    // UnreadRune causes the next call to ReadRune to return the sam
    UnreadRune() error
    // SkipSpace skips space in the input. Newlines are treated as s
    // unless the scan operation is Scanln, Fscanln or Sscanln, in w
    // a newline is treated as EOF.
    SkipSpace()
    // Token skips space in the input if skipSpace is true, then ret
    // run of Unicode code points c satisfying f(c). If f is nil,
    // !unicode.IsSpace(c) is used; that is, the token will hold non
    // characters. Newlines are treated as space unless the scan op
    // is Scanln, Fscanln or Sscanln, in which case a newline is tre
    // EOF. The returned slice points to shared data that may be ov
    // by the next call to Token, a call to a Scan function using th
    // as input, or when the calling Scan method returns.
    Token(skipSpace bool, f func(rune) bool) (token []byte, err erro
    // Width returns the value of the width option and whether it ha
    // The unit is Unicode code points.
    Width() (wid int, ok bool)
    // Because ReadRune is implemented by the interface, Read should
    // called by the scanning routines and a valid implementation of
    // ScanState may choose always to return an error from Read.
    Read(buf []byte) (n int, err error)
}
```

ScanState represents the scanner state passed to custom scanners. Scanners may do rune-at-a-time scanning or ask the ScanState to discover the next space-delimited token.

type [Scanner](#)

```
type Scanner interface {  
    Scan(state ScanState, verb rune) error  
}
```

Scanner is implemented by any value that has a Scan method, which scans the input for the representation of a value and stores the result in the receiver, which must be a pointer to be useful. The Scan method is called for any argument to Scan, Scanf, or Scanln that implements it.

type [State](#)

```
type State interface {
    // Write is the function to call to emit formatted output to be
    Write(b []byte) (ret int, err error)
    // Width returns the value of the width option and whether it has
    Width() (wid int, ok bool)
    // Precision returns the value of the precision option and whether
    Precision() (prec int, ok bool)

    // Flag returns whether the flag c, a character, has been set.
    Flag(c int) bool
}
```

State represents the printer state passed to custom formatters. It provides access to the `io.Writer` interface plus information about the flags and options for the operand's format specifier.

type [Stringer](#)

```
type Stringer interface {  
    String() string  
}
```

Stringer is implemented by any value that has a String method, which defines the “native” format for that value. The String method is used to print values passed as an operand to a %s or %v format or to an unformatted printer such as Print.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg/go

Name	Synopsis
ast	Package ast declares the types used to represent syntax trees for Go packages.
build	Package build gathers information about Go packages.
doc	Package doc extracts source code documentation from a Go AST.
parser	Package parser implements a parser for Go source files.
printer	Package printer implements printing of AST nodes.
scanner	Package scanner implements a scanner for Go source text.
token	Package token defines constants representing the lexical tokens of the Go programming language and basic operations on tokens (printing, predicates).

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package ast

```
import "go/ast"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package ast declares the types used to represent syntax trees for Go packages.

Index

[func FileExports\(src *File\) bool](#)
[func FilterDecl\(decl Decl, f Filter\) bool](#)
[func FilterFile\(src *File, f Filter\) bool](#)
[func FilterPackage\(pkg *Package, f Filter\) bool](#)
[func Fprint\(w io.Writer, fset *token.FileSet, x interface{}, f FieldFilter\) \(err error\)](#)
[func Inspect\(node Node, f func\(Node\) bool\)](#)
[func IsExported\(name string\) bool](#)
[func NotNilFilter\(_ string, v reflect.Value\) bool](#)
[func PackageExports\(pkg *Package\) bool](#)
[func Print\(fset *token.FileSet, x interface{}\) error](#)
[func SortImports\(fset *token.FileSet, f *File\)](#)
[func Walk\(v Visitor, node Node\)](#)
[type ArrayType](#)
 [func \(x *ArrayType\) End\(\) token.Pos](#)
 [func \(x *ArrayType\) Pos\(\) token.Pos](#)
[type AssignStmt](#)
 [func \(s *AssignStmt\) End\(\) token.Pos](#)
 [func \(s *AssignStmt\) Pos\(\) token.Pos](#)
[type BadDecl](#)
 [func \(d *BadDecl\) End\(\) token.Pos](#)
 [func \(d *BadDecl\) Pos\(\) token.Pos](#)
[type BadExpr](#)
 [func \(x *BadExpr\) End\(\) token.Pos](#)
 [func \(x *BadExpr\) Pos\(\) token.Pos](#)
[type BadStmt](#)
 [func \(s *BadStmt\) End\(\) token.Pos](#)
 [func \(s *BadStmt\) Pos\(\) token.Pos](#)
[type BasicLit](#)
 [func \(x *BasicLit\) End\(\) token.Pos](#)
 [func \(x *BasicLit\) Pos\(\) token.Pos](#)
[type BinaryExpr](#)
 [func \(x *BinaryExpr\) End\(\) token.Pos](#)
 [func \(x *BinaryExpr\) Pos\(\) token.Pos](#)
[type BlockStmt](#)

[func \(s *BlockStmt\) End\(\) token.Pos](#)
[func \(s *BlockStmt\) Pos\(\) token.Pos](#)
type BranchStmt
[func \(s *BranchStmt\) End\(\) token.Pos](#)
[func \(s *BranchStmt\) Pos\(\) token.Pos](#)
type CallExpr
[func \(x *CallExpr\) End\(\) token.Pos](#)
[func \(x *CallExpr\) Pos\(\) token.Pos](#)
type CaseClause
[func \(s *CaseClause\) End\(\) token.Pos](#)
[func \(s *CaseClause\) Pos\(\) token.Pos](#)
type ChanDir
type ChanType
[func \(x *ChanType\) End\(\) token.Pos](#)
[func \(x *ChanType\) Pos\(\) token.Pos](#)
type CommClause
[func \(s *CommClause\) End\(\) token.Pos](#)
[func \(s *CommClause\) Pos\(\) token.Pos](#)
type Comment
[func \(c *Comment\) End\(\) token.Pos](#)
[func \(c *Comment\) Pos\(\) token.Pos](#)
type CommentGroup
[func \(g *CommentGroup\) End\(\) token.Pos](#)
[func \(g *CommentGroup\) Pos\(\) token.Pos](#)
[func \(g *CommentGroup\) Text\(\) string](#)
type CompositeLit
[func \(x *CompositeLit\) End\(\) token.Pos](#)
[func \(x *CompositeLit\) Pos\(\) token.Pos](#)
type Decl
type DeclStmt
[func \(s *DeclStmt\) End\(\) token.Pos](#)
[func \(s *DeclStmt\) Pos\(\) token.Pos](#)
type DeferStmt
[func \(s *DeferStmt\) End\(\) token.Pos](#)
[func \(s *DeferStmt\) Pos\(\) token.Pos](#)
type Ellipsis
[func \(x *Ellipsis\) End\(\) token.Pos](#)
[func \(x *Ellipsis\) Pos\(\) token.Pos](#)
type EmptyStmt

func (s *EmptyStmt) End() token.Pos
func (s *EmptyStmt) Pos() token.Pos
type Expr
type ExprStmt
func (s *ExprStmt) End() token.Pos
func (s *ExprStmt) Pos() token.Pos
type Field
func (f *Field) End() token.Pos
func (f *Field) Pos() token.Pos
type FieldFilter
type FieldList
func (f *FieldList) End() token.Pos
func (f *FieldList) NumFields() int
func (f *FieldList) Pos() token.Pos
type File
func MergePackageFiles(pkg *Package, mode MergeMode) *File
func (f *File) End() token.Pos
func (f *File) Pos() token.Pos
type Filter
type ForStmt
func (s *ForStmt) End() token.Pos
func (s *ForStmt) Pos() token.Pos
type FuncDecl
func (d *FuncDecl) End() token.Pos
func (d *FuncDecl) Pos() token.Pos
type FuncLit
func (x *FuncLit) End() token.Pos
func (x *FuncLit) Pos() token.Pos
type FuncType
func (x *FuncType) End() token.Pos
func (x *FuncType) Pos() token.Pos
type GenDecl
func (d *GenDecl) End() token.Pos
func (d *GenDecl) Pos() token.Pos
type GoStmt
func (s *GoStmt) End() token.Pos
func (s *GoStmt) Pos() token.Pos
type Ident
func NewIdent(name string) *Ident

func (x *Ident) End() token.Pos
func (id *Ident) IsExported() bool
func (x *Ident) Pos() token.Pos
func (id *Ident) String() string
type IfStmt
func (s *IfStmt) End() token.Pos
func (s *IfStmt) Pos() token.Pos
type ImportSpec
func (s *ImportSpec) End() token.Pos
func (s *ImportSpec) Pos() token.Pos
type Importer
type IncDecStmt
func (s *IncDecStmt) End() token.Pos
func (s *IncDecStmt) Pos() token.Pos
type IndexExpr
func (x *IndexExpr) End() token.Pos
func (x *IndexExpr) Pos() token.Pos
type InterfaceType
func (x *InterfaceType) End() token.Pos
func (x *InterfaceType) Pos() token.Pos
type KeyValueExpr
func (x *KeyValueExpr) End() token.Pos
func (x *KeyValueExpr) Pos() token.Pos
type LabeledStmt
func (s *LabeledStmt) End() token.Pos
func (s *LabeledStmt) Pos() token.Pos
type MapType
func (x *MapType) End() token.Pos
func (x *MapType) Pos() token.Pos
type MergeMode
type Node
type ObjKind
func (kind ObjKind) String() string
type Object
func NewObj(kind ObjKind, name string) *Object
func (obj *Object) Pos() token.Pos
type Package
func NewPackage(fset *token.FileSet, files map[string]*File, importer
Importer, universe *Scope) (*Package, error)

[func \(p *Package\) End\(\) token.Pos](#)
[func \(p *Package\) Pos\(\) token.Pos](#)
type ParenExpr
[func \(x *ParenExpr\) End\(\) token.Pos](#)
[func \(x *ParenExpr\) Pos\(\) token.Pos](#)
type RangeStmt
[func \(s *RangeStmt\) End\(\) token.Pos](#)
[func \(s *RangeStmt\) Pos\(\) token.Pos](#)
type ReturnStmt
[func \(s *ReturnStmt\) End\(\) token.Pos](#)
[func \(s *ReturnStmt\) Pos\(\) token.Pos](#)
type Scope
[func NewScope\(outer *Scope\) *Scope](#)
[func \(s *Scope\) Insert\(obj *Object\) \(alt *Object\)](#)
[func \(s *Scope\) Lookup\(name string\) *Object](#)
[func \(s *Scope\) String\(\) string](#)
type SelectStmt
[func \(s *SelectStmt\) End\(\) token.Pos](#)
[func \(s *SelectStmt\) Pos\(\) token.Pos](#)
type SelectorExpr
[func \(x *SelectorExpr\) End\(\) token.Pos](#)
[func \(x *SelectorExpr\) Pos\(\) token.Pos](#)
type SendStmt
[func \(s *SendStmt\) End\(\) token.Pos](#)
[func \(s *SendStmt\) Pos\(\) token.Pos](#)
type SliceExpr
[func \(x *SliceExpr\) End\(\) token.Pos](#)
[func \(x *SliceExpr\) Pos\(\) token.Pos](#)
type Spec
type StarExpr
[func \(x *StarExpr\) End\(\) token.Pos](#)
[func \(x *StarExpr\) Pos\(\) token.Pos](#)
type Stmt
type StructType
[func \(x *StructType\) End\(\) token.Pos](#)
[func \(x *StructType\) Pos\(\) token.Pos](#)
type SwitchStmt
[func \(s *SwitchStmt\) End\(\) token.Pos](#)
[func \(s *SwitchStmt\) Pos\(\) token.Pos](#)

```
type TypeAssertExpr  
  func (x *TypeAssertExpr) End() token.Pos  
  func (x *TypeAssertExpr) Pos() token.Pos  
type TypeSpec  
  func (s *TypeSpec) End() token.Pos  
  func (s *TypeSpec) Pos() token.Pos  
type TypeSwitchStmt  
  func (s *TypeSwitchStmt) End() token.Pos  
  func (s *TypeSwitchStmt) Pos() token.Pos  
type UnaryExpr  
  func (x *UnaryExpr) End() token.Pos  
  func (x *UnaryExpr) Pos() token.Pos  
type ValueSpec  
  func (s *ValueSpec) End() token.Pos  
  func (s *ValueSpec) Pos() token.Pos  
type Visitor
```

Examples

[Inspect](#)
[Print](#)

Package files

[ast.go](#) [filter.go](#) [import.go](#) [print.go](#) [resolve.go](#) [scope.go](#) [walk.go](#)

func [FileExports](#)

```
func FileExports(src *File) bool
```

FileExports trims the AST for a Go source file in place such that only exported nodes remain: all top-level identifiers which are not exported and their associated information (such as type, initial value, or function body) are removed. Non-exported fields and methods of exported types are stripped. The File.Comments list is not changed.

FileExports returns true if there are exported declarations; it returns false otherwise.

func [FilterDecl](#)

```
func FilterDecl(decl Decl, f Filter) bool
```

FilterDecl trims the AST for a Go declaration in place by removing all names (including struct field and interface method names, but not from parameter lists) that don't pass through the filter f.

FilterDecl returns true if there are any declared names left after filtering; it returns false otherwise.

func [FilterFile](#)

```
func FilterFile(src *File, f Filter) bool
```

FilterFile trims the AST for a Go file in place by removing all names from top-level declarations (including struct field and interface method names, but not from parameter lists) that don't pass through the filter `f`. If the declaration is empty afterwards, the declaration is removed from the AST. The `File.Comments` list is not changed.

FilterFile returns true if there are any top-level declarations left after filtering; it returns false otherwise.

func [FilterPackage](#)

```
func FilterPackage(pkg *Package, f Filter) bool
```

FilterPackage trims the AST for a Go package in place by removing all names from top-level declarations (including struct field and interface method names, but not from parameter lists) that don't pass through the filter `f`. If the declaration is empty afterwards, the declaration is removed from the AST. The `pkg.Files` list is not changed, so that file names and top-level package comments don't get lost.

FilterPackage returns true if there are any top-level declarations left after filtering; it returns false otherwise.

func [Fprint](#)

```
func Fprint(w io.Writer, fset *token.FileSet, x interface{}, f Field
```

Fprint prints the (sub-)tree starting at AST node x to w. If fset != nil, position information is interpreted relative to that file set. Otherwise positions are printed as integer values (file set specific offsets).

A non-nil FieldFilter f may be provided to control the output: struct fields for which f(fieldname, fieldvalue) is true are printed; all others are filtered from the output.

func [Inspect](#)

```
func Inspect(node Node, f func(Node) bool)
```

Inspect traverses an AST in depth-first order: It starts by calling `f(node)`; `node` must not be `nil`. If `f` returns `true`, `Inspect` invokes `f` for all the non-`nil` children of `node`, recursively.

? Example

? Example

This example demonstrates how to inspect the AST of a Go program.

Code:

```
// src is the input for which we want to inspect the AST.
src := `
package p
const c = 1.0
var X = f(3.14)*2 + c
`

// Create the AST by parsing src.
fset := token.NewFileSet() // positions are relative to fset
f, err := parser.ParseFile(fset, "src.go", src, 0)
if err != nil {
    panic(err)
}

// Inspect the AST and print all identifiers and literals.
ast.Inspect(f, func(n ast.Node) bool {
    var s string
    switch x := n.(type) {
    case *ast.BasicLit:
        s = x.Value
    case *ast.Ident:
        s = x.Name
    }
    if s != "" {
        fmt.Printf("%s:\t%s\n", fset.Position(n.Pos()), s)
    }
    return true
})
```

Output:

```
src.go:2:9:      p
src.go:3:7:      c
src.go:3:11:     1.0
src.go:4:5:      X
src.go:4:9:      f
src.go:4:11:     3.14
src.go:4:17:     2
src.go:4:21:     c
```

func IsExported

```
func IsExported(name string) bool
```

IsExported returns whether name is an exported Go symbol (i.e., whether it begins with an uppercase letter).

func [NotNilFilter](#)

```
func NotNilFilter(_ string, v reflect.Value) bool
```

NotNilFilter returns true for field values that are not nil; it returns false otherwise.

func PackageExports

```
func PackageExports(pkg *Package) bool
```

PackageExports trims the AST for a Go package in place such that only exported nodes remain. The pkg.Files list is not changed, so that file names and top-level package comments don't get lost.

PackageExports returns true if there are exported declarations; it returns false otherwise.

func [Print](#)

```
func Print(fset *token.FileSet, x interface{}) error
```

Print prints x to standard output, skipping nil fields. Print(fset, x) is the same as Fprint(os.Stdout, fset, x, NotNilFilter).

? Example

? Example

This example shows what an AST looks like when printed for debugging.

Code:

```
// src is the input for which we want to print the AST.
src := `
package main
func main() {
println("Hello, World!")
}
`

// Create the AST by parsing src.
fset := token.NewFileSet() // positions are relative to fset
f, err := parser.ParseFile(fset, "", src, 0)
if err != nil {
    panic(err)
}

// Print the AST.
ast.Print(fset, f)
```

Output:

```
0 *ast.File {
  1 . Package: 2:1
  2 . Name: *ast.Ident {
  3 . . NamePos: 2:9
  4 . . Name: "main"
  5 . }
  6 . Decls: []ast.Decl (len = 1) {
  7 . . 0: *ast.FuncDecl {
```

```

 8 . . . Name: *ast.Ident {
 9 . . . . NamePos: 3:6
10 . . . . Name: "main"
11 . . . . Obj: *ast.Object {
12 . . . . . Kind: func
13 . . . . . Name: "main"
14 . . . . . Decl: *(obj @ 7)
15 . . . . }
16 . . . }
17 . . . Type: *ast.FuncType {
18 . . . . Func: 3:1
19 . . . . Params: *ast.FieldList {
20 . . . . . Opening: 3:10
21 . . . . . Closing: 3:11
22 . . . . }
23 . . . }
24 . . . Body: *ast.BlockStmt {
25 . . . . Lbrace: 3:13
26 . . . . List: []ast.Stmt (len = 1) {
27 . . . . . 0: *ast.ExprStmt {
28 . . . . . . X: *ast.CallExpr {
29 . . . . . . . Fun: *ast.Ident {
30 . . . . . . . . NamePos: 4:2
31 . . . . . . . . Name: "println"
32 . . . . . . . }
33 . . . . . . . Lparen: 4:9
34 . . . . . . . Args: []ast.Expr (len = 1) {
35 . . . . . . . . 0: *ast.BasicLit {
36 . . . . . . . . . ValuePos: 4:10
37 . . . . . . . . . Kind: STRING
38 . . . . . . . . . Value: "\"Hello, World!\""
39 . . . . . . . . }
40 . . . . . . . }
41 . . . . . . . Ellipsis: -
42 . . . . . . . Rparen: 4:25
43 . . . . . . }
44 . . . . . }
45 . . . . }
46 . . . . Rbrace: 5:1
47 . . . }
48 . . }
49 . }
50 . Scope: *ast.Scope {
51 . . Objects: map[string]*ast.Object (len = 1) {
52 . . . "main": *(obj @ 11)
53 . . }
54 . }
55 . Unresolved: []*ast.Ident (len = 1) {
56 . . 0: *(obj @ 29)

```

57 . }
58 }

func [SortImports](#)

```
func SortImports(fset *token.FileSet, f *File)
```

SortImports sorts runs of consecutive import lines in import blocks in f.

func [Walk](#)

```
func Walk(v Visitor, node Node)
```

Walk traverses an AST in depth-first order: It starts by calling `v.Visit(node)`; `node` must not be `nil`. If the visitor `w` returned by `v.Visit(node)` is not `nil`, Walk is invoked recursively with visitor `w` for each of the non-`nil` children of `node`, followed by a call of `w.Visit(nil)`.

type [ArrayType](#)

```
type ArrayType struct {
    Lbrack token.Pos // position of "["
    Len    Expr          // Ellipsis node for [...]T array types, nil fo
    Elt    Expr          // element type
}
```

An ArrayType node represents an array or slice type.

func (***ArrayType**) [End](#)

```
func (x *ArrayType) End() token.Pos
```

func (***ArrayType**) [Pos](#)

```
func (x *ArrayType) Pos() token.Pos
```

type [AssignStmt](#)

```
type AssignStmt struct {
    Lhs    []Expr
    TokPos token.Pos // position of Tok
    Tok    token.Token // assignment token, DEFINE
    Rhs    []Expr
}
```

An AssignStmt node represents an assignment or a short variable declaration.

func (*AssignStmt) [End](#)

```
func (s *AssignStmt) End() token.Pos
```

func (*AssignStmt) [Pos](#)

```
func (s *AssignStmt) Pos() token.Pos
```

type [BadDecl](#)

```
type BadDecl struct {  
    From, To token.Pos // position range of bad declaration  
}
```

A `BadDecl` node is a placeholder for declarations containing syntax errors for which no correct declaration nodes can be created.

func (*BadDecl) [End](#)

```
func (d *BadDecl) End() token.Pos
```

func (*BadDecl) [Pos](#)

```
func (d *BadDecl) Pos() token.Pos
```

Pos and End implementations for declaration nodes.

type BadExpr

```
type BadExpr struct {  
    From, To token.Pos // position range of bad expression  
}
```

A `BadExpr` node is a placeholder for expressions containing syntax errors for which no correct expression nodes can be created.

func (*BadExpr) End

```
func (x *BadExpr) End() token.Pos
```

func (*BadExpr) Pos

```
func (x *BadExpr) Pos() token.Pos
```

Pos and End implementations for expression/type nodes.

type [BadStmt](#)

```
type BadStmt struct {  
    From, To token.Pos // position range of bad statement  
}
```

A `BadStmt` node is a placeholder for statements containing syntax errors for which no correct statement nodes can be created.

func (*BadStmt) [End](#)

```
func (s *BadStmt) End() token.Pos
```

func (*BadStmt) [Pos](#)

```
func (s *BadStmt) Pos() token.Pos
```

`Pos` and `End` implementations for statement nodes.

type **BasicLit**

```
type BasicLit struct {
    ValuePos token.Pos // literal position
    Kind     token.Token // token.INT, token.FLOAT, token.IMAG, toke
    Value    string    // literal string; e.g. 42, 0x7f, 3.14, 1e-
}
```

A BasicLit node represents a literal of basic type.

func (*BasicLit) **End**

```
func (x *BasicLit) End() token.Pos
```

func (*BasicLit) **Pos**

```
func (x *BasicLit) Pos() token.Pos
```

type **BinaryExpr**

```
type BinaryExpr struct {
    X      Expr      // left operand
    OpPos  token.Pos  // position of Op
    Op     token.Token // operator
    Y      Expr      // right operand
}
```

A BinaryExpr node represents a binary expression.

func (*BinaryExpr) **End**

```
func (x *BinaryExpr) End() token.Pos
```

func (*BinaryExpr) **Pos**

```
func (x *BinaryExpr) Pos() token.Pos
```

type BlockStmt

```
type BlockStmt struct {
    Lbrace token.Pos // position of "{"
    List   []Stmt
    Rbrace token.Pos // position of "}"
}
```

A BlockStmt node represents a braced statement list.

func (*BlockStmt) End

```
func (s *BlockStmt) End() token.Pos
```

func (*BlockStmt) Pos

```
func (s *BlockStmt) Pos() token.Pos
```

type [BranchStmt](#)

```
type BranchStmt struct {  
    TokPos token.Pos // position of Tok  
    Tok    token.Token // keyword token (BREAK, CONTINUE, GOTO, FALL  
    Label  *Ident      // label name; or nil  
}
```

A BranchStmt node represents a break, continue, goto, or fallthrough statement.

func (*BranchStmt) [End](#)

```
func (s *BranchStmt) End() token.Pos
```

func (*BranchStmt) [Pos](#)

```
func (s *BranchStmt) Pos() token.Pos
```

type [CallExpr](#)

```
type CallExpr struct {
    Fun      Expr      // function expression
    Lparen   token.Pos // position of "("
    Args     []Expr   // function arguments; or nil
    Ellipsis token.Pos // position of "...", if any
    Rparen   token.Pos // position of ")"
}
```

A CallExpr node represents an expression followed by an argument list.

func (*CallExpr) [End](#)

```
func (x *CallExpr) End() token.Pos
```

func (*CallExpr) [Pos](#)

```
func (x *CallExpr) Pos() token.Pos
```

type [CaseClause](#)

```
type CaseClause struct {
    Case token.Pos // position of "case" or "default" keyword
    List []Expr    // list of expressions or types; nil means default
    Colon token.Pos // position of ":"
    Body []Stmt     // statement list; or nil
}
```

A CaseClause represents a case of an expression or type switch statement.

func (***CaseClause**) [End](#)

```
func (s *CaseClause) End() token.Pos
```

func (***CaseClause**) [Pos](#)

```
func (s *CaseClause) Pos() token.Pos
```

type [ChanDir](#)

```
type ChanDir int
```

The direction of a channel type is indicated by one of the following constants.

```
const (  
    SEND ChanDir = 1 << iota  
    RECV  
)
```

type [ChanType](#)

```
type ChanType struct {  
    Begin token.Pos // position of "chan" keyword or "<-" (whichever  
    Dir    ChanDir   // channel direction  
    Value Expr       // value type  
}
```

A ChanType node represents a channel type.

func (***ChanType**) [End](#)

```
func (x *ChanType) End() token.Pos
```

func (***ChanType**) [Pos](#)

```
func (x *ChanType) Pos() token.Pos
```

type CommClause

```
type CommClause struct {
    Case token.Pos // position of "case" or "default" keyword
    Comm Stmt      // send or receive statement; nil means default
    Colon token.Pos // position of ":"
    Body []Stmt     // statement list; or nil
}
```

A CommClause node represents a case of a select statement.

func (*CommClause) End

```
func (s *CommClause) End() token.Pos
```

func (*CommClause) Pos

```
func (s *CommClause) Pos() token.Pos
```

type Comment

```
type Comment struct {
    Slash token.Pos // position of "/" starting the comment
    Text  string    // comment text (excluding '\n' for //-style com
}
```

A Comment node represents a single //-style or /*-style comment.

func (*Comment) End

```
func (c *Comment) End() token.Pos
```

func (*Comment) Pos

```
func (c *Comment) Pos() token.Pos
```

type [CommentGroup](#)

```
type CommentGroup struct {  
    List []*Comment // len(List) > 0  
}
```

A CommentGroup represents a sequence of comments with no other tokens and no empty lines between.

func (*CommentGroup) [End](#)

```
func (g *CommentGroup) End() token.Pos
```

func (*CommentGroup) [Pos](#)

```
func (g *CommentGroup) Pos() token.Pos
```

func (*CommentGroup) [Text](#)

```
func (g *CommentGroup) Text() string
```

Text returns the text of the comment, with the comment markers - //, /*, and */ - removed.

type CompositeLit

```
type CompositeLit struct {
    Type   Expr      // literal type; or nil
    Lbrace token.Pos // position of "{"
    Elts   []Expr     // list of composite elements; or nil
    Rbrace token.Pos // position of "}"
}
```

A CompositeLit node represents a composite literal.

func (*CompositeLit) End

```
func (x *CompositeLit) End() token.Pos
```

func (*CompositeLit) Pos

```
func (x *CompositeLit) Pos() token.Pos
```

type Decl

```
type Decl interface {  
    Node  
    // contains filtered or unexported methods  
}
```

All declaration nodes implement the Decl interface.

type DeclStmt

```
type DeclStmt struct {  
    Decl Decl  
}
```

A DeclStmt node represents a declaration in a statement list.

func (*DeclStmt) End

```
func (s *DeclStmt) End() token.Pos
```

func (*DeclStmt) Pos

```
func (s *DeclStmt) Pos() token.Pos
```

type **DeferStmt**

```
type DeferStmt struct {  
    Defer token.Pos // position of "defer" keyword  
    Call *CallExpr  
}
```

A DeferStmt node represents a defer statement.

func (*DeferStmt) **End**

```
func (s *DeferStmt) End() token.Pos
```

func (*DeferStmt) **Pos**

```
func (s *DeferStmt) Pos() token.Pos
```

type [Ellipsis](#)

```
type Ellipsis struct {  
    Ellipsis token.Pos // position of "..."  
    Elt      Expr      // ellipsis element type (parameter lists onl  
}
```

An Ellipsis node stands for the "..." type in a parameter list or the "..." length in an array type.

func (*Ellipsis) [End](#)

```
func (x *Ellipsis) End() token.Pos
```

func (*Ellipsis) [Pos](#)

```
func (x *Ellipsis) Pos() token.Pos
```

type EmptyStmt

```
type EmptyStmt struct {  
    Semicolon token.Pos // position of preceding ";"  
}
```

An EmptyStmt node represents an empty statement. The "position" of the empty statement is the position of the immediately preceding semicolon.

func (*EmptyStmt) End

```
func (s *EmptyStmt) End() token.Pos
```

func (*EmptyStmt) Pos

```
func (s *EmptyStmt) Pos() token.Pos
```

type [Expr](#)

```
type Expr interface {  
    Node  
    // contains filtered or unexported methods  
}
```

All expression nodes implement the Expr interface.

type ExprStmt

```
type ExprStmt struct {  
    X Expr // expression  
}
```

An ExprStmt node represents a (stand-alone) expression in a statement list.

func (*ExprStmt) End

```
func (s *ExprStmt) End() token.Pos
```

func (*ExprStmt) Pos

```
func (s *ExprStmt) Pos() token.Pos
```

type [Field](#)

```
type Field struct {
    Doc      *CommentGroup // associated documentation; or nil
    Names    []*Ident       // field/method/parameter names; or nil if
    Type     Expr        // field/method/parameter type
    Tag      *BasicLit     // field tag; or nil
    Comment  *CommentGroup // line comments; or nil
}
```

A Field represents a Field declaration list in a struct type, a method list in an interface type, or a parameter/result declaration in a signature.

func (*Field) [End](#)

```
func (f *Field) End() token.Pos
```

func (*Field) [Pos](#)

```
func (f *Field) Pos() token.Pos
```

type [FieldFilter](#)

```
type FieldFilter func(name string, value reflect.Value) bool
```

A FieldFilter may be provided to Fprint to control the output.

type [FieldList](#)

```
type FieldList struct {  
    Opening token.Pos // position of opening parenthesis/brace, if a  
    List    []*Field // field list; or nil  
    Closing token.Pos // position of closing parenthesis/brace, if a  
}
```

A FieldList represents a list of Fields, enclosed by parentheses or braces.

func (*FieldList) [End](#)

```
func (f *FieldList) End() token.Pos
```

func (*FieldList) [NumFields](#)

```
func (f *FieldList) NumFields() int
```

NumFields returns the number of (named and anonymous fields) in a FieldList.

func (*FieldList) [Pos](#)

```
func (f *FieldList) Pos() token.Pos
```

type [File](#)

```
type File struct {
    Doc          *CommentGroup // associated documentation; or nil
    Package      token.Pos      // position of "package" keyword
    Name         *Ident         // package name
    Decls        []Decl        // top-level declarations; or nil
    Scope        *Scope        // package scope (this file only)
    Imports      []*ImportSpec // imports in this file
    Unresolved   []*Ident       // unresolved identifiers in this file
    Comments     []*CommentGroup // list of all comments in the source
}
```

A File node represents a Go source file.

The Comments list contains all comments in the source file in order of appearance, including the comments that are pointed to from other nodes via Doc and Comment fields.

func [MergePackageFiles](#)

```
func MergePackageFiles(pkg *Package, mode MergeMode) *File
```

MergePackageFiles creates a file AST by merging the ASTs of the files belonging to a package. The mode flags control merging behavior.

func (*File) [End](#)

```
func (f *File) End() token.Pos
```

func (*File) [Pos](#)

```
func (f *File) Pos() token.Pos
```

type **Filter**

```
type Filter func(string) bool
```

type **ForStmt**

```
type ForStmt struct {
    For token.Pos // position of "for" keyword
    Init Stmt     // initialization statement; or nil
    Cond Expr     // condition; or nil
    Post Stmt     // post iteration statement; or nil
    Body *BlockStmt
}
```

A ForStmt represents a for statement.

func (*ForStmt) **End**

```
func (s *ForStmt) End() token.Pos
```

func (*ForStmt) **Pos**

```
func (s *ForStmt) Pos() token.Pos
```

type [FuncDecl](#)

```
type FuncDecl struct {  
    Doc    *CommentGroup // associated documentation; or nil  
    Recv   *FieldList    // receiver (methods); or nil (functions)  
    Name   *Ident         // function/method name  
    Type   *FuncType     // position of Func keyword, parameters and r  
    Body   *BlockStmt    // function body; or nil (forward declaration  
}
```

A FuncDecl node represents a function declaration.

func (*FuncDecl) [End](#)

```
func (d *FuncDecl) End() token.Pos
```

func (*FuncDecl) [Pos](#)

```
func (d *FuncDecl) Pos() token.Pos
```

type FuncLit

```
type FuncLit struct {  
    Type *FuncType // function type  
    Body *BlockStmt // function body  
}
```

A FuncLit node represents a function literal.

func (*FuncLit) End

```
func (x *FuncLit) End() token.Pos
```

func (*FuncLit) Pos

```
func (x *FuncLit) Pos() token.Pos
```

type **FuncType**

```
type FuncType struct {  
    Func    token.Pos // position of "func" keyword  
    Params  *FieldList // (incoming) parameters; or nil  
    Results *FieldList // (outgoing) results; or nil  
}
```

A FuncType node represents a function type.

func (*FuncType) End

```
func (x *FuncType) End() token.Pos
```

func (*FuncType) Pos

```
func (x *FuncType) Pos() token.Pos
```

type [GenDecl](#)

```
type GenDecl struct {
    Doc      *CommentGroup // associated documentation; or nil
    TokPos   token.Pos    // position of Tok
    Tok      token.Token   // IMPORT, CONST, TYPE, VAR
    Lparen   token.Pos    // position of '(', if any
    Specs    []Spec
    Rparen   token.Pos    // position of ')', if any
}
```

A GenDecl node (generic declaration node) represents an import, constant, type or variable declaration. A valid Lparen position (Lparen.Line > 0) indicates a parenthesized declaration.

Relationship between Tok value and Specs element type:

```
token.IMPORT  *ImportSpec
token.CONST   *ValueSpec
token.TYPE    *TypeSpec
token.VAR     *ValueSpec
```

func (*GenDecl) [End](#)

```
func (d *GenDecl) End() token.Pos
```

func (*GenDecl) [Pos](#)

```
func (d *GenDecl) Pos() token.Pos
```

type GoStmt

```
type GoStmt struct {  
    Go    token.Pos // position of "go" keyword  
    Call *CallExpr  
}
```

A GoStmt node represents a go statement.

func (*GoStmt) End

```
func (s *GoStmt) End() token.Pos
```

func (*GoStmt) Pos

```
func (s *GoStmt) Pos() token.Pos
```

type [Ident](#)

```
type Ident struct {
    NamePos token.Pos // identifier position
    Name    string    // identifier name
    Obj     *Object   // denoted object; or nil
}
```

An Ident node represents an identifier.

func [NewIdent](#)

```
func NewIdent(name string) *Ident
```

NewIdent creates a new Ident without position. Useful for ASTs generated by code other than the Go parser.

func (***Ident**) [End](#)

```
func (x *Ident) End() token.Pos
```

func (***Ident**) [IsExported](#)

```
func (id *Ident) IsExported() bool
```

IsExported returns whether id is an exported Go symbol (i.e., whether it begins with an uppercase letter).

func (***Ident**) [Pos](#)

```
func (x *Ident) Pos() token.Pos
```

func (***Ident**) [String](#)

```
func (id *Ident) String() string
```

type [IfStmt](#)

```
type IfStmt struct {
    If    token.Pos // position of "if" keyword
    Init Stmt       // initialization statement; or nil
    Cond Expr       // condition
    Body *BlockStmt
    Else Stmt // else branch; or nil
}
```

An IfStmt node represents an if statement.

func (*IfStmt) [End](#)

```
func (s *IfStmt) End() token.Pos
```

func (*IfStmt) [Pos](#)

```
func (s *IfStmt) Pos() token.Pos
```

type [ImportSpec](#)

```
type ImportSpec struct {
    Doc      *CommentGroup // associated documentation; or nil
    Name     *Ident         // local package name (including "."); or
    Path     *BasicLit     // import path
    Comment  *CommentGroup // line comments; or nil
    EndPos   token.Pos    // end of spec (overrides Path.Pos if nonz
}
```

An ImportSpec node represents a single package import.

func (*ImportSpec) [End](#)

```
func (s *ImportSpec) End() token.Pos
```

func (*ImportSpec) [Pos](#)

```
func (s *ImportSpec) Pos() token.Pos
```

Pos and End implementations for spec nodes.

type Importer

```
type Importer func(imports map[string]*Object, path string) (pkg *Ob
```

An Importer resolves import paths to package Objects. The imports map records the packages already imported, indexed by package id (canonical import path).

An Importer must determine the canonical import path and check the map to see if it is already present in the imports map. If so, the Importer can return the map entry. Otherwise, the Importer should load the package data for the given path into a new *Object (pkg), record pkg in the imports map, and then return pkg.

type IncDecStmt

```
type IncDecStmt struct {  
    X      Expr  
    TokPos token.Pos // position of Tok  
    Tok    token.Token // INC or DEC  
}
```

An IncDecStmt node represents an increment or decrement statement.

func (*IncDecStmt) End

```
func (s *IncDecStmt) End() token.Pos
```

func (*IncDecStmt) Pos

```
func (s *IncDecStmt) Pos() token.Pos
```

type [IndexExpr](#)

```
type IndexExpr struct {
    X      Expr      // expression
    Lbrack token.Pos // position of "["
    Index  Expr      // index expression
    Rbrack token.Pos // position of "]"
}
```

An IndexExpr node represents an expression followed by an index.

func (*IndexExpr) [End](#)

```
func (x *IndexExpr) End() token.Pos
```

func (*IndexExpr) [Pos](#)

```
func (x *IndexExpr) Pos() token.Pos
```

type [InterfaceType](#)

```
type InterfaceType struct {  
    Interface token.Pos // position of "interface" keyword  
    Methods   *FieldList // list of methods  
    Incomplete bool      // true if (source) methods are missing in  
}
```

An InterfaceType node represents an interface type.

func (*InterfaceType) [End](#)

```
func (x *InterfaceType) End() token.Pos
```

func (*InterfaceType) [Pos](#)

```
func (x *InterfaceType) Pos() token.Pos
```

type KeyValueExpr

```
type KeyValueExpr struct {
    Key    Expr
    Colon token.Pos // position of ":"
    Value Expr
}
```

A KeyValueExpr node represents (key : value) pairs in composite literals.

func (*KeyValueExpr) End

```
func (x *KeyValueExpr) End() token.Pos
```

func (*KeyValueExpr) Pos

```
func (x *KeyValueExpr) Pos() token.Pos
```

type LabeledStmt

```
type LabeledStmt struct {  
    Label *Ident  
    Colon token.Pos // position of ":"  
    Stmt  Stmt  
}
```

A LabeledStmt node represents a labeled statement.

func (*LabeledStmt) End

```
func (s *LabeledStmt) End() token.Pos
```

func (*LabeledStmt) Pos

```
func (s *LabeledStmt) Pos() token.Pos
```

type [MapType](#)

```
type MapType struct {  
    Map    token.Pos // position of "map" keyword  
    Key    Expr  
    Value  Expr  
}
```

A MapType node represents a map type.

func (***MapType**) [End](#)

```
func (x *MapType) End() token.Pos
```

func (***MapType**) [Pos](#)

```
func (x *MapType) Pos() token.Pos
```

type [MergeMode](#)

```
type MergeMode uint
```

The MergeMode flags control the behavior of MergePackageFiles.

```
const (  
    // If set, duplicate function declarations are excluded.  
    FilterFuncDuplicates MergeMode = 1 << iota  
    // If set, comments that are not associated with a specific  
    // AST node (as Doc or Comment) are excluded.  
    FilterUnassociatedComments  
    // If set, duplicate import declarations are excluded.  
    FilterImportDuplicates  
)
```

type [Node](#)

```
type Node interface {  
    Pos() token.Pos // position of first character belonging to the  
    End() token.Pos // position of first character immediately after  
}
```

All node types implement the Node interface.

type [ObjKind](#)

```
type ObjKind int
```

ObjKind describes what an object represents.

```
const (  
    Bad ObjKind = iota // for error handling  
    Pkg           // package  
    Con           // constant  
    Typ           // type  
    Var           // variable  
    Fun           // function or method  
    Lbl           // label  
)
```

The list of possible Object kinds.

func (ObjKind) [String](#)

```
func (kind ObjKind) String() string
```

type [Object](#)

```
type Object struct {
    Kind ObjKind
    Name string      // declared name
    Decl interface{} // corresponding Field, XxxSpec, FuncDecl, Label
    Data interface{} // object-specific data; or nil
    Type interface{} // place holder for type information; may be nil
}
```

An Object describes a named language entity such as a package, constant, type, variable, function (incl. methods), or label.

The Data field contains object-specific data:

Kind	Data type	Data value
Pkg	*Scope	package scope
Con	int	iota for the respective declaration
Con	!= nil	constant value

func [NewObj](#)

```
func NewObj(kind ObjKind, name string) *Object
```

NewObj creates a new object of a given kind and name.

func (*Object) [Pos](#)

```
func (obj *Object) Pos() token.Pos
```

Pos computes the source position of the declaration of an object name. The result may be an invalid position if it cannot be computed (obj.Decl may be nil or not correct).

type [Package](#)

```
type Package struct {
    Name      string           // package name
    Scope     *Scope          // package scope across all files
    Imports   map[string]*Object // map of package id -> package object
    Files     map[string]*File  // Go source files by filename
}
```

A Package node represents a set of source files collectively building a Go package.

func [NewPackage](#)

```
func NewPackage(fset *token.FileSet, files map[string]*File, importer
```

`NewPackage` creates a new Package node from a set of File nodes. It resolves unresolved identifiers across files and updates each file's Unresolved list accordingly. If a non-nil importer and universe scope are provided, they are used to resolve identifiers not declared in any of the package files. Any remaining unresolved identifiers are reported as undeclared. If the files belong to different packages, one package name is selected and files with different package names are reported and then ignored. The result is a package node and a `scanner.ErrorList` if there were errors.

func (*Package) [End](#)

```
func (p *Package) End() token.Pos
```

func (*Package) [Pos](#)

```
func (p *Package) Pos() token.Pos
```

type ParenExpr

```
type ParenExpr struct {
    Lparen token.Pos // position of "("
    X      Expr      // parenthesized expression
    Rparen token.Pos // position of ")"
}
```

A ParenExpr node represents a parenthesized expression.

func (*ParenExpr) End

```
func (x *ParenExpr) End() token.Pos
```

func (*ParenExpr) Pos

```
func (x *ParenExpr) Pos() token.Pos
```

type [RangeStmt](#)

```
type RangeStmt struct {
    For      token.Pos // position of "for" keyword
    Key, Value Expr    // Value may be nil
    TokPos   token.Pos // position of Tok
    Tok      token.Token // ASSIGN, DEFINE
    X        Expr    // value to range over
    Body     *BlockStmt
}
```

A RangeStmt represents a for statement with a range clause.

func (*RangeStmt) [End](#)

```
func (s *RangeStmt) End() token.Pos
```

func (*RangeStmt) [Pos](#)

```
func (s *RangeStmt) Pos() token.Pos
```

type ReturnStmt

```
type ReturnStmt struct {  
    Return token.Pos // position of "return" keyword  
    Results []Expr    // result expressions; or nil  
}
```

A ReturnStmt node represents a return statement.

func (*ReturnStmt) End

```
func (s *ReturnStmt) End() token.Pos
```

func (*ReturnStmt) Pos

```
func (s *ReturnStmt) Pos() token.Pos
```

type [Scope](#)

```
type Scope struct {
    Outer    *Scope
    Objects map[string]*Object
}
```

A Scope maintains the set of named language entities declared in the scope and a link to the immediately surrounding (outer) scope.

func [NewScope](#)

```
func NewScope(outer *Scope) *Scope
```

NewScope creates a new scope nested in the outer scope.

func (***Scope**) [Insert](#)

```
func (s *Scope) Insert(obj *Object) (alt *Object)
```

Insert attempts to insert a named object obj into the scope s. If the scope already contains an object alt with the same name, Insert leaves the scope unchanged and returns alt. Otherwise it inserts obj and returns nil."

func (***Scope**) [Lookup](#)

```
func (s *Scope) Lookup(name string) *Object
```

Lookup returns the object with the given name if it is found in scope s, otherwise it returns nil. Outer scopes are ignored.

func (***Scope**) [String](#)

```
func (s *Scope) String() string
```

Debugging support

type [SelectStmt](#)

```
type SelectStmt struct {  
    Select token.Pos // position of "select" keyword  
    Body   *BlockStmt // CommClauses only  
}
```

An SelectStmt node represents a select statement.

func (*SelectStmt) [End](#)

```
func (s *SelectStmt) End() token.Pos
```

func (*SelectStmt) [Pos](#)

```
func (s *SelectStmt) Pos() token.Pos
```

type SelectorExpr

```
type SelectorExpr struct {  
    X Expr // expression  
    Sel *Ident // field selector  
}
```

A SelectorExpr node represents an expression followed by a selector.

func (*SelectorExpr) End

```
func (x *SelectorExpr) End() token.Pos
```

func (*SelectorExpr) Pos

```
func (x *SelectorExpr) Pos() token.Pos
```

type [SendStmt](#)

```
type SendStmt struct {  
    Chan Expr  
    Arrow token.Pos // position of "<-"  
    Value Expr  
}
```

A SendStmt node represents a send statement.

func (*SendStmt) [End](#)

```
func (s *SendStmt) End() token.Pos
```

func (*SendStmt) [Pos](#)

```
func (s *SendStmt) Pos() token.Pos
```

type [SliceExpr](#)

```
type SliceExpr struct {
    X      Expr    // expression
    Lbrack token.Pos // position of "["
    Low    Expr    // begin of slice range; or nil
    High   Expr    // end of slice range; or nil
    Rbrack token.Pos // position of "]"
}
```

An SliceExpr node represents an expression followed by slice indices.

func (*SliceExpr) [End](#)

```
func (x *SliceExpr) End() token.Pos
```

func (*SliceExpr) [Pos](#)

```
func (x *SliceExpr) Pos() token.Pos
```

type [Spec](#)

```
type Spec interface {  
    Node  
    // contains filtered or unexported methods  
}
```

The Spec type stands for any of *ImportSpec, *ValueSpec, and *TypeSpec.

type StarExpr

```
type StarExpr struct {  
    Star token.Pos // position of "*"   
    X    Expr      // operand   
}
```

A StarExpr node represents an expression of the form "*" Expression. Semantically it could be a unary "*" expression, or a pointer type.

func (*StarExpr) End

```
func (x *StarExpr) End() token.Pos
```

func (*StarExpr) Pos

```
func (x *StarExpr) Pos() token.Pos
```

type [Stmt](#)

```
type Stmt interface {  
    Node  
    // contains filtered or unexported methods  
}
```

All statement nodes implement the Stmt interface.

type [StructType](#)

```
type StructType struct {  
    Struct      token.Pos // position of "struct" keyword  
    Fields      *FieldList // list of field declarations  
    Incomplete bool      // true if (source) fields are missing in  
}
```

A StructType node represents a struct type.

func (***StructType**) [End](#)

```
func (x *StructType) End() token.Pos
```

func (***StructType**) [Pos](#)

```
func (x *StructType) Pos() token.Pos
```

type [SwitchStmt](#)

```
type SwitchStmt struct {
    Switch token.Pos // position of "switch" keyword
    Init   Stmt       // initialization statement; or nil
    Tag    Expr       // tag expression; or nil
    Body   *BlockStmt // CaseClauses only
}
```

A SwitchStmt node represents an expression switch statement.

func (*SwitchStmt) [End](#)

```
func (s *SwitchStmt) End() token.Pos
```

func (*SwitchStmt) [Pos](#)

```
func (s *SwitchStmt) Pos() token.Pos
```

type TypeAssertExpr

```
type TypeAssertExpr struct {  
    X      Expr // expression  
    Type Expr // asserted type; nil means type switch X.(type)  
}
```

A TypeAssertExpr node represents an expression followed by a type assertion.

func (*TypeAssertExpr) End

```
func (x *TypeAssertExpr) End() token.Pos
```

func (*TypeAssertExpr) Pos

```
func (x *TypeAssertExpr) Pos() token.Pos
```

type [TypeSpec](#)

```
type TypeSpec struct {
    Doc      *CommentGroup // associated documentation; or nil
    Name     *Ident         // type name
    Type     Expr          // *Ident, *ParenExpr, *SelectorExpr, *Sta
    Comment  *CommentGroup // line comments; or nil
}
```

A TypeSpec node represents a type declaration (TypeSpec production).

func (*TypeSpec) [End](#)

```
func (s *TypeSpec) End() token.Pos
```

func (*TypeSpec) [Pos](#)

```
func (s *TypeSpec) Pos() token.Pos
```

type [TypeSwitchStmt](#)

```
type TypeSwitchStmt struct {  
    Switch token.Pos // position of "switch" keyword  
    Init  Stmt        // initialization statement; or nil  
    Assign Stmt       // x := y.(type) or y.(type)  
    Body  *BlockStmt // CaseClauses only  
}
```

An TypeSwitchStmt node represents a type switch statement.

func (*TypeSwitchStmt) [End](#)

```
func (s *TypeSwitchStmt) End() token.Pos
```

func (*TypeSwitchStmt) [Pos](#)

```
func (s *TypeSwitchStmt) Pos() token.Pos
```

type UnaryExpr

```
type UnaryExpr struct {
    OpPos token.Pos // position of Op
    Op    token.Token // operator
    X     Expr      // operand
}
```

A UnaryExpr node represents a unary expression. Unary "*" expressions are represented via StarExpr nodes.

func (*UnaryExpr) End

```
func (x *UnaryExpr) End() token.Pos
```

func (*UnaryExpr) Pos

```
func (x *UnaryExpr) Pos() token.Pos
```

type [ValueSpec](#)

```
type ValueSpec struct {
    Doc      *CommentGroup // associated documentation; or nil
    Names    []*Ident      // value names (len(Names) > 0)
    Type     Expr       // value type; or nil
    Values   []Expr       // initial values; or nil
    Comment  *CommentGroup // line comments; or nil
}
```

A ValueSpec node represents a constant or variable declaration (ConstSpec or VarSpec production).

func (*ValueSpec) [End](#)

```
func (s *ValueSpec) End() token.Pos
```

func (*ValueSpec) [Pos](#)

```
func (s *ValueSpec) Pos() token.Pos
```

type [Visitor](#)

```
type Visitor interface {  
    Visit(node Node) (w Visitor)  
}
```

A Visitor's Visit method is invoked for each node encountered by Walk. If the result visitor w is not nil, Walk visits each of the children of node with the visitor w, followed by a call of w.Visit(nil).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package build

```
import "go/build"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package build gathers information about Go packages.

Go Path

The Go path is a list of directory trees containing Go source code. It is consulted to resolve imports that cannot be found in the standard Go tree. The default path is the value of the GOPATH environment variable, interpreted as a path list appropriate to the operating system (on Unix, the variable is a colon-separated string; on Windows, a semicolon-separated string; on Plan 9, a list).

Each directory listed in the Go path must have a prescribed structure:

The src/ directory holds source code. The path below 'src' determines the import path or executable name.

The pkg/ directory holds installed package objects. As in the Go tree, each target operating system and architecture pair has its own subdirectory of pkg (pkg/GOOS_GOARCH).

If DIR is a directory listed in the Go path, a package with source in DIR/src/foo/bar can be imported as "foo/bar" and has its compiled form installed to "DIR/pkg/GOOS_GOARCH/foo/bar.a" (or, for gccgo, "DIR/pkg/gccgo/foo/libbar.a").

The bin/ directory holds compiled commands. Each command is named for its source directory, but only using the final element, not the entire path. That is, the command with source in DIR/src/foo/quux is installed into DIR/bin/quux, not DIR/bin/foo/quux. The foo/ is stripped so that you can add DIR/bin to your PATH to get at the installed commands.

Here's an example directory layout:

```
GOPATH=/home/user/gocode  
  
/home/user/gocode/  
  src/  
    foo/
```

```

        bar/                (go code in package bar)
            x.go
        quux/              (go code in package main)
            y.go
bin/
  quux                    (installed command)
pkg/
  linux_amd64/
    foo/
      bar.a                (installed package object)

```

Build Constraints

A build constraint is a line comment beginning with the directive `+build` that lists the conditions under which a file should be included in the package. Constraints may appear in any kind of source file (not just Go), but they must appear near the top of the file, preceded only by blank lines and other line comments.

A build constraint is evaluated as the OR of space-separated options; each option evaluates as the AND of its comma-separated terms; and each term is an alphanumeric word or, preceded by `!`, its negation. That is, the build constraint:

```
// +build linux,386 darwin,!cgo
```

corresponds to the boolean formula:

```
(linux AND 386) OR (darwin AND (NOT cgo))
```

During a particular build, the following words are satisfied:

- the target operating system, as spelled by `runtime.GOOS`
- the target architecture, as spelled by `runtime.GOARCH`
- "cgo", if `ctxt.CgoEnabled` is true
- any additional words listed in `ctxt.BuildTags`

If a file's name, after stripping the extension and a possible `_test` suffix, matches `*_GOOS`, `*_GOARCH`, or `*_GOOS_GOARCH` for any known operating system and architecture values, then the file is considered to have an implicit build constraint requiring those terms.

To keep a file from being considered for the build:

```
// +build ignore
```

(any other unsatisfied word will work as well, but “ignore” is conventional.)

To build a file only when using cgo, and only on Linux and OS X:

```
// +build linux,cgo darwin,cgo
```

Such a file is usually paired with another file implementing the default functionality for other systems, which in this case would carry the constraint:

```
// +build !linux !darwin !cgo
```

Naming a file `dns_windows.go` will cause it to be included only when building the package for Windows; similarly, `math_386.s` will be included only when building the package for 32-bit x86.

Index

Variables

[func ArchChar\(goarch string\) \(string, error\)](#)

[func IsLocalImport\(path string\) bool](#)

[type Context](#)

[func \(ctxt *Context\) Import\(path string, srcDir string, mode ImportMode\) \(*Package, error\)](#)

[func \(ctxt *Context\) ImportDir\(dir string, mode ImportMode\) \(*Package, error\)](#)

[func \(ctxt *Context\) SrcDirs\(\) \[\]string](#)

[type ImportMode](#)

[type NoGoError](#)

[func \(e *NoGoError\) Error\(\) string](#)

[type Package](#)

[func Import\(path, srcDir string, mode ImportMode\) \(*Package, error\)](#)

[func ImportDir\(dir string, mode ImportMode\) \(*Package, error\)](#)

[func \(p *Package\) IsCommand\(\) bool](#)

Package files

[build.go](#) [doc.go](#) [syslist.go](#)

Variables

```
var ToolDir = filepath.Join(runtime.GOROOT(), "pkg/tool/"+runtime.GO
```

ToolDir is the directory containing build tools.

func [ArchChar](#)

```
func ArchChar(goarch string) (string, error)
```

ArchChar returns the architecture character for the given goarch. For example, ArchChar("amd64") returns "6".

func [IsLocalImport](#)

```
func IsLocalImport(path string) bool
```

IsLocalImport reports whether the import path is a local import path, like ".", "..", "./foo", or "../foo".

type [Context](#)

```
type Context struct {
    GOARCH      string // target architecture
    GOOS        string // target operating system
    GOROOT      string // Go root
    GOPATH      string // Go path
    CgoEnabled  bool   // whether cgo can be used
    BuildTags   []string // additional tags to recognize in +build l
    UseAllFiles bool   // use files regardless of +build lines, fi
    Compiler    string

    // JoinPath joins the sequence of path fragments into a single p
    // If JoinPath is nil, Import uses filepath.Join.
    JoinPath func(elem ...string) string

    // SplitPathList splits the path list into a slice of individual
    // If SplitPathList is nil, Import uses filepath.SplitList.
    SplitPathList func(list string) []string

    // IsAbsPath reports whether path is an absolute path.
    // If IsAbsPath is nil, Import uses filepath.IsAbs.
    IsAbsPath func(path string) bool

    // IsDir reports whether the path names a directory.
    // If IsDir is nil, Import calls os.Stat and uses the result's I
    IsDir func(path string) bool

    // HasSubdir reports whether dir is a subdirectory of
    // (perhaps multiple levels below) root.
    // If so, HasSubdir sets rel to a slash-separated path that
    // can be joined to root to produce a path equivalent to dir.
    // If HasSubdir is nil, Import uses an implementation built on
    // filepath.EvalSymlinks.
    HasSubdir func(root, dir string) (rel string, ok bool)

    // ReadDir returns a slice of os.FileInfo, sorted by Name,
    // describing the content of the named directory.
    // If ReadDir is nil, Import uses io.ReadDir.
    ReadDir func(dir string) (fi []os.FileInfo, err error)

    // OpenFile opens a file (not a directory) for reading.
    // If OpenFile is nil, Import uses os.Open.
    OpenFile func(path string) (r io.ReadCloser, err error)
}
```

A Context specifies the supporting context for a build.

```
var Default Context = defaultContext()
```

Default is the default Context for builds. It uses the GOARCH, GOOS, GOROOT, and GOPATH environment variables if set, or else the compiled code's GOARCH, GOOS, and GOROOT.

func (*Context) [Import](#)

```
func (ctxt *Context) Import(path string, srcDir string, mode ImportM
```

Import returns details about the Go package named by the import path, interpreting local import paths relative to the srcDir directory. If the path is a local import path naming a package that can be imported using a standard import path, the returned package will set p.ImportPath to that path.

In the directory containing the package, .go, .c, .h, and .s files are considered part of the package except for:

- .go files in package documentation
- files starting with _ or . (likely editor temporary files)
- files with build constraints not satisfied by the context

If an error occurs, Import returns a non-nil error also returns a non-nil *Package containing partial information.

func (*Context) [ImportDir](#)

```
func (ctxt *Context) ImportDir(dir string, mode ImportMode) (*Packag
```

ImportDir is like Import but processes the Go package found in the named directory.

func (*Context) [SrcDirs](#)

```
func (ctxt *Context) SrcDirs() []string
```

SrcDirs returns a list of package source root directories. It draws from the current Go root and Go path but omits directories that do not exist.

type [ImportMode](#)

```
type ImportMode uint
```

An ImportMode controls the behavior of the Import method.

```
const (  
    // If FindOnly is set, Import stops after locating the directory  
    // that should contain the sources for a package. It does not  
    // read any files in the directory.  
    FindOnly ImportMode = 1 << iota  
  
    // If AllowBinary is set, Import can be satisfied by a compiled  
    // package object without corresponding sources.  
    AllowBinary  
)
```

type [NoGoError](#)

```
type NoGoError struct {  
    Dir string  
}
```

NoGoError is the error used by Import to describe a directory containing no Go source files.

func (*NoGoError) [Error](#)

```
func (e *NoGoError) Error() string
```

type [Package](#)

```
type Package struct {
    Dir      string // directory containing package sources
    Name     string // package name
    Doc      string // documentation synopsis
    ImportPath string // import path of package (" " if unknown)
    Root     string // root of Go tree where this package lives
    SrcRoot  string // package source root directory (" " if unknown)
    PkgRoot  string // package install root directory (" " if unknown)
    BinDir   string // command install directory (" " if unknown)
    Goroot   bool   // package found in Go root
    PkgObj   string

    // Source files
    GoFiles  []string // .go source files (excluding CgoFiles, Test
    CgoFiles []string // .go source files that import "C"
    CFiles   []string // .c source files
    HFiles   []string // .h source files
    SFiles   []string // .s source files
    SysoFiles []string

    // Cgo directives
    CgoPkgConfig []string // Cgo pkg-config directives
    CgoCFLAGS    []string // Cgo CFLAGS directives
    CgoLDFLAGS   []string

    // Dependency information
    Imports  []string // imports from GoFiles, CgoFiles
    ImportPos map[string][]token.Position

    // Test information
    TestGoFiles  []string // _test.go files in
    TestImports  []string // imports from TestG
    TestImportPos map[string][]token.Position // line information f
    XTestGoFiles []string // _test.go files out
    XTestImports []string // imports from XTest
    XTestImportPos map[string][]token.Position // line information f
}
```

A Package describes the Go package found in a directory.

func [Import](#)

```
func Import(path, srcDir string, mode ImportMode) (*Package, error)
```

Import is shorthand for Default.Import.

func [ImportDir](#)

```
func ImportDir(dir string, mode ImportMode) (*Package, error)
```

ImportDir is shorthand for Default.ImportDir.

func (*Package) [IsCommand](#)

```
func (p *Package) IsCommand() bool
```

IsCommand reports whether the package is considered a command to be installed (not just a library). Packages named "main" are treated as commands.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package doc

```
import "go/doc"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package doc extracts source code documentation from a Go AST.

Index

[func Examples\(files ...*ast.File\) \[\]*Example](#)

[func Synopsis\(s string\) string](#)

[func ToHTML\(w io.Writer, text string, words map\[string\]string\)](#)

[func ToText\(w io.Writer, text string, indent, preIndent string, width int\)](#)

[type Example](#)

[type Filter](#)

[type Func](#)

[type Mode](#)

[type Package](#)

[func New\(pkg *ast.Package, importPath string, mode Mode\) *Package](#)

[func \(p *Package\) Filter\(f Filter\)](#)

[type Type](#)

[type Value](#)

Package files

[comment.go](#) [doc.go](#) [example.go](#) [exports.go](#) [filter.go](#) [reader.go](#) [synopsis.go](#)

func [Examples](#)

```
func Examples(files ...*ast.File) []*Example
```

func Synopsis

```
func Synopsis(s string) string
```

Synopsis returns a cleaned version of the first sentence in s. That sentence ends after the first period followed by space and not preceded by exactly one uppercase letter. The result string has no \n, \r, or \t characters and uses only single spaces between words.

func [ToHTML](#)

```
func ToHTML(w io.Writer, text string, words map[string]string)
```

ToHTML converts comment text to formatted HTML. The comment was prepared by DocReader, so it is known not to have leading, trailing blank lines nor to have trailing spaces at the end of lines. The comment markers have already been removed.

Turn each run of multiple `\n` into `</p><p>`. Turn each run of indented lines into a `<pre>` block without indent. Enclose headings with header tags.

URLs in the comment text are converted into links; if the URL also appears in the words map, the link is taken from the map (if the corresponding map value is the empty string, the URL is not converted into a link).

Go identifiers that appear in the words map are italicized; if the corresponding map value is not the empty string, it is considered a URL and the word is converted into a link.

func [ToText](#)

```
func ToText(w io.Writer, text string, indent, preIndent string, width int)
```

ToText prepares comment text for presentation in textual output. It wraps paragraphs of text to width or fewer Unicode code points and then prefixes each line with the indent. In preformatted sections (such as program text), it prefixes each non-blank line with preIndent.

type [Example](#)

```
type Example struct {  
    Name      string // name of the item being exemplified  
    Doc       string // example function doc string  
    Code      ast.Node  
    Comments  []*ast.CommentGroup  
    Output    string // expected output  
}
```

type **Filter**

```
type Filter func(string) bool
```

type [Func](#)

```
type Func struct {
    Doc string
    Name string
    Decl *ast.FuncDecl

    // methods
    // (for functions, these fields have the respective zero value)
    Recv string // actual receiver "T" or "*T"
    Orig string // original receiver "T" or "*T"
    Level int    // embedding level; 0 means not embedded
}
```

Func is the documentation for a func declaration.

type [Mode](#)

```
type Mode int
```

Mode values control the operation of New.

```
const (  
    // extract documentation for all package-level declarations,  
    // not just exported ones  
    AllDecls Mode = 1 << iota  
  
    // show all embedded methods, not just the ones of  
    // invisible (unexported) anonymous fields  
    AllMethods  
)
```

type [Package](#)

```
type Package struct {
    Doc      string
    Name     string
    ImportPath string
    Imports  []string
    Filenames []string
    Bugs     []string

    // declarations
    Consts []*Value
    Types []*Type
    Vars  []*Value
    Funcs []*Func
}
```

Package is the documentation for an entire package.

func [New](#)

```
func New(pkg *ast.Package, importPath string, mode Mode) *Package
```

New computes the package documentation for the given package AST. New takes ownership of the AST pkg and may edit or overwrite it.

func (*Package) [Filter](#)

```
func (p *Package) Filter(f Filter)
```

Filter eliminates documentation for names that don't pass through the filter f. TODO: Recognize "Type.Method" as a name.

type [Type](#)

```
type Type struct {
    Doc string
    Name string
    Decl *ast.GenDecl

    // associated declarations
    Consts []*Value // sorted list of constants of (mostly) this ty
    Vars    []*Value // sorted list of variables of (mostly) this ty
    Funcs  []*Func  // sorted list of functions returning this type
    Methods []*Func  // sorted list of methods (including embedded c
}
```

Type is the documentation for a type declaration.

type [Value](#)

```
type Value struct {
    Doc    string
    Names []string // var or const names in declaration order
    Decl  *ast.GenDecl
        // contains filtered or unexported fields
}
```

Value is the documentation for a (possibly grouped) var or const declaration.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package parser

```
import "go/parser"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package parser implements a parser for Go source files. Input may be provided in a variety of forms (see the various Parse* functions); the output is an abstract syntax tree (AST) representing the Go source. The parser is invoked through one of the Parse* functions.

Index

[func ParseDir\(fset *token.FileSet, path string, filter func\(os.FileInfo\) bool, mode Mode\) \(pkgs map\[string\]*ast.Package, first error\)](#)

[func ParseExpr\(x string\) \(ast.Expr, error\)](#)

[func ParseFile\(fset *token.FileSet, filename string, src interface{}, mode Mode\) \(*ast.File, error\)](#)

[type Mode](#)

Examples

[ParseFile](#)

Package files

[interface.go](#) [parser.go](#)

func ParseDir

```
func ParseDir(fset *token.FileSet, path string, filter func(os.FileI
```

ParseDir calls ParseFile for the files in the directory specified by path and returns a map of package name -> package AST with all the packages found. If filter != nil, only the files with os.FileInfo entries passing through the filter are considered. The mode bits are passed to ParseFile unchanged. Position information is recorded in the file set fset.

If the directory couldn't be read, a nil map and the respective error are returned. If a parse error occurred, a non-nil but incomplete map and the first error encountered are returned.

func ParseExpr

```
func ParseExpr(x string) (ast.Expr, error)
```

ParseExpr is a convenience function for obtaining the AST of an expression x. The position information recorded in the AST is undefined.

func [ParseFile](#)

```
func ParseFile(fset *token.FileSet, filename string, src interface{})
```

ParseFile parses the source code of a single Go source file and returns the corresponding ast.File node. The source code may be provided via the filename of the source file, or via the src parameter.

If src != nil, ParseFile parses the source from src and the filename is only used when recording position information. The type of the argument for the src parameter must be string, []byte, or io.Reader. If src == nil, ParseFile parses the file specified by filename.

The mode parameter controls the amount of source text parsed and other optional parser functionality. Position information is recorded in the file set fset.

If the source couldn't be read, the returned AST is nil and the error indicates the specific failure. If the source was read but syntax errors were found, the result is a partial AST (with ast.Bad* nodes representing the fragments of erroneous source code). Multiple errors are returned via a scanner.ErrorList which is sorted by file position.

? Example

? Example

Code:

```
fset := token.NewFileSet() // positions are relative to fset

// Parse the file containing this very example
// but stop after processing the imports.
f, err := parser.ParseFile(fset, "example_test.go", nil, parser.Impo
if err != nil {
    fmt.Println(err)
    return
}

// Print the imports from the file's AST.
for _, s := range f.Imports {
    fmt.Println(s.Path.Value)
```

```
}
```

Output:

```
"fmt"  
"go/parser"  
"go/token"
```

type [Mode](#)

```
type Mode uint
```

A Mode value is a set of flags (or 0). They control the amount of source code parsed and other optional parser functionality.

```
const (  
    PackageClauseOnly Mode = 1 << iota // parsing stops after packag  
    ImportsOnly           // parsing stops after import  
    ParseComments        // parse comments and add the  
    Trace                 // print a trace of parsed pr  
    DeclarationErrors    // report declaration errors  
    SpuriousErrors       // report all (not just the f  
)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package printer

```
import "go/printer"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package printer implements printing of AST nodes.

Index

[func Fprint\(output io.Writer, fset *token.FileSet, node interface{ }\) error](#)

[type CommentedNode](#)

[type Config](#)

[func \(cfg *Config\) Fprint\(output io.Writer, fset *token.FileSet, node](#)

[interface{ }\) error](#)

[type Mode](#)

Examples

[Fprint](#)

Package files

[nodes.go printer.go](#)

func [Fprint](#)

```
func Fprint(output io.Writer, fset *token.FileSet, node interface{})
```

Fprint "pretty-prints" an AST node to output. It calls Config.Fprint with default settings.

? Example

? Example

Code:

```
// Parse source file and extract the AST without comments for
// this function, with position information referring to the
// file set fset.
funcAST, fset := parseFunc("example_test.go", "ExampleFprint")

// Print the function body into buffer buf.
// The file set is provided to the printer so that it knows
// about the original source formatting and can add additional
// line breaks where they were present in the source.
var buf bytes.Buffer
printer.Fprint(&buf, fset, funcAST.Body)

// Remove braces {} enclosing the function body, unindent,
// and trim leading and trailing white space.
s := buf.String()
s = s[1 : len(s)-1]
s = strings.TrimSpace(strings.Replace(s, "\n\t", "\n", -1))

// Print the cleaned-up body text to stdout.
fmt.Println(s)
```

Output:

```
funcAST, fset := parseFunc("example_test.go", "ExampleFprint")

var buf bytes.Buffer
printer.Fprint(&buf, fset, funcAST.Body)

s := buf.String()
s = s[1 : len(s)-1]
s = strings.TrimSpace(strings.Replace(s, "\n\t", "\n", -1))
```

```
fmt.Println(s)
```

type [CommentedNode](#)

```
type CommentedNode struct {  
    Node      interface{} // *ast.File, or ast.Expr, ast.Decl, ast.Sp  
    Comments []*ast.CommentGroup  
}
```

A `CommentedNode` bundles an AST node and corresponding comments. It may be provided as argument to any of the `Fprint` functions.

type [Config](#)

```
type Config struct {  
    Mode      Mode // default: 0  
    Tabwidth  int  // default: 8  
}
```

A Config node controls the output of Fprint.

func (***Config**) [Fprint](#)

```
func (cfg *Config) Fprint(output io.Writer, fset *token.FileSet, nod
```

Fprint "pretty-prints" an AST node to output for a given configuration cfg. Position information is interpreted relative to the file set fset. The node type must be *ast.File, *CommentedNode, or assignment-compatible to ast.Expr, ast.Decl, ast.Spec, or ast.Stmt.

type [Mode](#)

```
type Mode uint
```

A Mode value is a set of flags (or 0). They control printing.

```
const (  
    RawFormat Mode = 1 << iota // do not use a tabwriter; if set, Us  
    TabIndent           // use tabs for indentation independe  
    UseSpaces          // use spaces instead of tabs for ali  
    SourcePos          // emit //line comments to preserve c  
)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package scanner

```
import "go/scanner"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package scanner implements a scanner for Go source text. It takes a []byte as source which can then be tokenized through repeated calls to the Scan method.

Index

[func PrintError\(w io.Writer, err error\)](#)

[type Error](#)

[func \(e Error\) Error\(\) string](#)

[type ErrorHandler](#)

[type ErrorList](#)

[func \(p *ErrorList\) Add\(pos token.Position, msg string\)](#)

[func \(p ErrorList\) Err\(\) error](#)

[func \(p ErrorList\) Error\(\) string](#)

[func \(p ErrorList\) Len\(\) int](#)

[func \(p ErrorList\) Less\(i, j int\) bool](#)

[func \(p *ErrorList\) RemoveMultiples\(\)](#)

[func \(p *ErrorList\) Reset\(\)](#)

[func \(p ErrorList\) Sort\(\)](#)

[func \(p ErrorList\) Swap\(i, j int\)](#)

[type Mode](#)

[type Scanner](#)

[func \(s *Scanner\) Init\(file *token.File, src \[\]byte, err ErrorHandler, mode Mode\)](#)

[func \(s *Scanner\) Scan\(\) \(pos token.Pos, tok token.Token, lit string\)](#)

Examples

[Scanner.Scan](#)

Package files

[errors.go scanner.go](#)

func [PrintError](#)

```
func PrintError(w io.Writer, err error)
```

PrintError is a utility function that prints a list of errors to w, one error per line, if the err parameter is an ErrorList. Otherwise it prints the err string.

type [Error](#)

```
type Error struct {  
    Pos token.Position  
    Msg string  
}
```

In an `ErrorList`, an error is represented by an `*Error`. The position `Pos`, if valid, points to the beginning of the offending token, and the error condition is described by `Msg`.

func (Error) [Error](#)

```
func (e Error) Error() string
```

`Error` implements the error interface.

type ErrorHandler

```
type ErrorHandler func(pos token.Position, msg string)
```

An ErrorHandler may be provided to Scanner.Init. If a syntax error is encountered and a handler was installed, the handler is called with a position and an error message. The position points to the beginning of the offending token.

type [ErrorList](#)

```
type ErrorList []*Error
```

ErrorList is a list of *Errors. The zero value for an ErrorList is an empty ErrorList ready to use.

func (*ErrorList) [Add](#)

```
func (p *ErrorList) Add(pos token.Position, msg string)
```

Add adds an Error with given position and error message to an ErrorList.

func (ErrorList) [Err](#)

```
func (p ErrorList) Err() error
```

Err returns an error equivalent to this error list. If the list is empty, Err returns nil.

func (ErrorList) [Error](#)

```
func (p ErrorList) Error() string
```

An ErrorList implements the error interface.

func (ErrorList) [Len](#)

```
func (p ErrorList) Len() int
```

ErrorList implements the sort Interface.

func (ErrorList) [Less](#)

```
func (p ErrorList) Less(i, j int) bool
```

func (*ErrorList) [RemoveMultiples](#)

```
func (p *ErrorList) RemoveMultiples()
```

RemoveMultiples sorts an ErrorList and removes all but the first error per line.

func (*ErrorList) [Reset](#)

```
func (p *ErrorList) Reset()
```

Reset resets an ErrorList to no errors.

func (ErrorList) [Sort](#)

```
func (p ErrorList) Sort()
```

Sort sorts an ErrorList. *Error entries are sorted by position, other errors are sorted by error message, and before any *Error entry.

func (ErrorList) [Swap](#)

```
func (p ErrorList) Swap(i, j int)
```

type [Mode](#)

```
type Mode uint
```

A mode value is set of flags (or 0). They control scanner behavior.

```
const (  
    ScanComments Mode = 1 << iota // return comments as COMMENT token  
)
```

type [Scanner](#)

```
type Scanner struct {  
    // public state - ok to modify  
    ErrorCount int // number of errors encountered  
    // contains filtered or unexported fields  
}
```

A Scanner holds the scanner's internal state while processing a given text. It can be allocated as part of another data structure but must be initialized via `Init` before use.

func (*Scanner) [Init](#)

```
func (s *Scanner) Init(file *token.File, src []byte, err ErrorHandler)
```

`Init` prepares the scanner `s` to tokenize the text `src` by setting the scanner at the beginning of `src`. The scanner uses the file set `file` for position information and it adds line information for each line. It is ok to re-use the same file when re-scanning the same file as line information which is already present is ignored. `Init` causes a panic if the file size does not match the `src` size.

Calls to `Scan` will invoke the error handler `err` if they encounter a syntax error and `err` is not nil. Also, for each error encountered, the Scanner field `ErrorCount` is incremented by one. The `mode` parameter determines how comments are handled.

Note that `Init` may call `err` if there is an error in the first character of the file.

func (*Scanner) [Scan](#)

```
func (s *Scanner) Scan() (pos token.Pos, tok token.Token, lit string)
```

`Scan` scans the next token and returns the token position, the token, and its literal string if applicable. The source end is indicated by `token.EOF`.

If the returned token is a literal (`token.IDENT`, `token.INT`, `token.FLOAT`, `token.IMAG`, `token.CHAR`, `token.STRING`) or `token.COMMENT`, the literal

string has the corresponding value.

If the returned token is `token.SEMICOLON`, the corresponding literal string is ";" if the semicolon was present in the source, and "\n" if the semicolon was inserted because of a newline or at EOF.

If the returned token is `token.ILLEGAL`, the literal string is the offending character.

In all other cases, `Scan` returns an empty literal string.

For more tolerant parsing, `Scan` will return a valid token if possible even if a syntax error was encountered. Thus, even if the resulting token sequence contains no illegal tokens, a client may not assume that no error occurred. Instead it must check the scanner's `ErrorCount` or the number of calls of the error handler, if there was one installed.

`Scan` adds line information to the file added to the file set with `Init`. Token positions are relative to that file and thus relative to the file set.

? Example

? Example

Code:

```
// src is the input that we want to tokenize.
src := []byte("cos(x) + 1i*sin(x) // Euler")

// Initialize the scanner.
var s scanner.Scanner
fset := token.NewFileSet() // positions are relative
file := fset.AddFile("", fset.Base(), len(src)) // register input "file"
s.Init(file, src, nil /* no error handler */, scanner.ScanComments)

// Repeated calls to Scan yield the token sequence found in the input
for {
    pos, tok, lit := s.Scan()
    if tok == token.EOF {
        break
    }
    fmt.Printf("%s\t%s\t%q\n", fset.Position(pos), tok, lit)
}
```

Output:

```
1:1      IDENT  "cos"  
1:4      (      ""  
1:5      IDENT  "x"  
1:6      )      ""  
1:8      +      ""  
1:10     IMAG  "1i"  
1:12     *      ""  
1:13     IDENT  "sin"  
1:16     (      ""  
1:17     IDENT  "x"  
1:18     )      ""  
1:20     ;      "\n"  
1:20     COMMENT "// Euler"
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package token

```
import "go/token"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package token defines constants representing the lexical tokens of the Go programming language and basic operations on tokens (printing, predicates).

Index

Constants

type File

[func \(f *File\) AddLine\(offset int\)](#)
[func \(f *File\) AddLineInfo\(offset int, filename string, line int\)](#)
[func \(f *File\) Base\(\) int](#)
[func \(f *File\) Line\(p Pos\) int](#)
[func \(f *File\) LineCount\(\) int](#)
[func \(f *File\) Name\(\) string](#)
[func \(f *File\) Offset\(p Pos\) int](#)
[func \(f *File\) Pos\(offset int\) Pos](#)
[func \(f *File\) Position\(p Pos\) \(pos Position\)](#)
[func \(f *File\) SetLines\(lines \[\]int\) bool](#)
[func \(f *File\) SetLinesForContent\(content \[\]byte\)](#)
[func \(f *File\) Size\(\) int](#)

type FileSet

[func NewFileSet\(\) *FileSet](#)
[func \(s *FileSet\) AddFile\(filename string, base, size int\) *File](#)
[func \(s *FileSet\) Base\(\) int](#)
[func \(s *FileSet\) File\(p Pos\) \(f *File\)](#)
[func \(s *FileSet\) Iterate\(f func\(*File\) bool\)](#)
[func \(s *FileSet\) Position\(p Pos\) \(pos Position\)](#)
[func \(s *FileSet\) Read\(decode func\(interface{ }\) error\) error](#)
[func \(s *FileSet\) Write\(encode func\(interface{ }\) error\) error](#)

type Pos

[func \(p Pos\) IsValid\(\) bool](#)

type Position

[func \(pos *Position\) IsValid\(\) bool](#)
[func \(pos Position\) String\(\) string](#)

type Token

[func Lookup\(ident string\) Token](#)
[func \(tok Token\) IsKeyword\(\) bool](#)
[func \(tok Token\) IsLiteral\(\) bool](#)
[func \(tok Token\) IsOperator\(\) bool](#)
[func \(op Token\) Precedence\(\) int](#)
[func \(tok Token\) String\(\) string](#)

Package files

[position.go](#) [serialize.go](#) [token.go](#)

Constants

```
const (  
    LowestPrec  = 0 // non-operators  
    UnaryPrec   = 6  
    HighestPrec = 7  
)
```

A set of constants for precedence-based expression parsing. Non-operators have lowest precedence, followed by operators starting with precedence 1 up to unary operators. The highest precedence corresponds serves as "catch-all" precedence for selector, indexing, and other operator and delimiter tokens.

type [File](#)

```
type File struct {  
    // contains filtered or unexported fields  
}
```

A File is a handle for a file belonging to a FileSet. A File has a name, size, and line offset table.

func (*File) [AddLine](#)

```
func (f *File) AddLine(offset int)
```

AddLine adds the line offset for a new line. The line offset must be larger than the offset for the previous line and smaller than the file size; otherwise the line offset is ignored.

func (*File) [AddLineInfo](#)

```
func (f *File) AddLineInfo(offset int, filename string, line int)
```

AddLineInfo adds alternative file and line number information for a given file offset. The offset must be larger than the offset for the previously added alternative line info and smaller than the file size; otherwise the information is ignored.

AddLineInfo is typically used to register alternative position information for //line filename:line comments in source files.

func (*File) [Base](#)

```
func (f *File) Base() int
```

Base returns the base offset of file f as registered with AddFile.

func (*File) [Line](#)

```
func (f *File) Line(p Pos) int
```

Line returns the line number for the given file position p; p must be a Pos value in that file or NoPos.

func (*File) [LineCount](#)

```
func (f *File) LineCount() int
```

LineCount returns the number of lines in file f.

func (*File) [Name](#)

```
func (f *File) Name() string
```

Name returns the file name of file f as registered with AddFile.

func (*File) [Offset](#)

```
func (f *File) Offset(p Pos) int
```

Offset returns the offset for the given file position p; p must be a valid Pos value in that file. `f.Offset(f.Pos(offset)) == offset`.

func (*File) [Pos](#)

```
func (f *File) Pos(offset int) Pos
```

Pos returns the Pos value for the given file offset; the offset must be `<= f.Size()`. `f.Pos(f.Offset(p)) == p`.

func (*File) [Position](#)

```
func (f *File) Position(p Pos) (pos Position)
```

Position returns the Position value for the given file position p; p must be a Pos value in that file or NoPos.

func (*File) [SetLines](#)

```
func (f *File) SetLines(lines []int) bool
```

SetLines sets the line offsets for a file and returns true if successful. The line offsets are the offsets of the first character of each line; for instance for the content "ab\nc\n" the line offsets are {0, 3}. An empty file has an empty line offset table. Each line offset must be larger than the offset for the previous line and smaller than the file size; otherwise SetLines fails and returns false.

func (*File) [SetLinesForContent](#)

```
func (f *File) SetLinesForContent(content []byte)
```

SetLinesForContent sets the line offsets for the given file content.

func (*File) [Size](#)

```
func (f *File) Size() int
```

Size returns the size of file f as registered with AddFile.

type [FileSet](#)

```
type FileSet struct {  
    // contains filtered or unexported fields  
}
```

A FileSet represents a set of source files. Methods of file sets are synchronized; multiple goroutines may invoke them concurrently.

func [NewFileSet](#)

```
func NewFileSet() *FileSet
```

NewFileSet creates a new file set.

func (*FileSet) [AddFile](#)

```
func (s *FileSet) AddFile(filename string, base, size int) *File
```

AddFile adds a new file with a given filename, base offset, and file size to the file set s and returns the file. Multiple files may have the same name. The base offset must not be smaller than the FileSet's Base(), and size must not be negative.

Adding the file will set the file set's Base() value to base + size + 1 as the minimum base value for the next file. The following relationship exists between a Pos value p for a given file offset offs:

$$\text{int}(p) = \text{base} + \text{offs}$$

with offs in the range [0, size] and thus p in the range [base, base+size]. For convenience, File.Pos may be used to create file-specific position values from a file offset.

func (*FileSet) [Base](#)

```
func (s *FileSet) Base() int
```

Base returns the minimum base offset that must be provided to AddFile when

adding the next file.

func (*FileSet) [File](#)

```
func (s *FileSet) File(p Pos) (f *File)
```

File returns the file that contains the position p. If no such file is found (for instance for p == NoPos), the result is nil.

func (*FileSet) [Iterate](#)

```
func (s *FileSet) Iterate(f func(*File) bool)
```

Iterate calls f for the files in the file set in the order they were added until f returns false.

func (*FileSet) [Position](#)

```
func (s *FileSet) Position(p Pos) (pos Position)
```

Position converts a Pos in the fileset into a general Position.

func (*FileSet) [Read](#)

```
func (s *FileSet) Read(decode func(interface{}) error) error
```

Read calls decode to deserialize a file set into s; s must not be nil.

func (*FileSet) [Write](#)

```
func (s *FileSet) Write(encode func(interface{}) error) error
```

Write calls encode to serialize the file set s.

type [Pos](#)

```
type Pos int
```

Pos is a compact encoding of a source position within a file set. It can be converted into a Position for a more convenient, but much larger, representation.

The Pos value for a given file is a number in the range [base, base+size], where base and size are specified when adding the file to the file set via AddFile.

To create the Pos value for a specific source offset, first add the respective file to the current file set (via FileSet.AddFile) and then call File.Pos(offset) for that file. Given a Pos value p for a specific file set fset, the corresponding Position value is obtained by calling fset.Position(p).

Pos values can be compared directly with the usual comparison operators: If two Pos values p and q are in the same file, comparing p and q is equivalent to comparing the respective source file offsets. If p and q are in different files, $p < q$ is true if the file implied by p was added to the respective file set before the file implied by q.

```
const NoPos Pos = 0
```

The zero value for Pos is NoPos; there is no file and line information associated with it, and NoPos().IsValid() is false. NoPos is always smaller than any other Pos value. The corresponding Position value for NoPos is the zero value for Position.

func (Pos) [IsValid](#)

```
func (p Pos) IsValid() bool
```

IsValid returns true if the position is valid.

type [Position](#)

```
type Position struct {  
    Filename string // filename, if any  
    Offset    int     // offset, starting at 0  
    Line      int     // line number, starting at 1  
    Column    int     // column number, starting at 1 (character count  
}
```

Position describes an arbitrary source position including the file, line, and column location. A Position is valid if the line number is > 0 .

func (*Position) [IsValid](#)

```
func (pos *Position) IsValid() bool
```

IsValid returns true if the position is valid.

func (Position) [String](#)

```
func (pos Position) String() string
```

String returns a string in one of several forms:

file:line:column	valid position with file name
line:column	valid position without file name
file	invalid position with file name
-	invalid position without file name

type [Token](#)

type Token int

Token is the set of lexical tokens of the Go programming language.

```
const (  
    // Special tokens  
    ILLEGAL Token = iota  
    EOF  
    COMMENT  
  
    // Identifiers and basic type literals  
    // (these tokens stand for classes of literals)  
    IDENT // main  
    INT   // 12345  
    FLOAT // 123.45  
    IMAG  // 123.45i  
    CHAR  // 'a'  
    STRING  
  
    // Operators and delimiters  
    ADD // +  
    SUB // -  
    MUL // *  
    QUO // /  
    REM // %  
  
    AND      // &  
    OR       // |  
    XOR      // ^  
    SHL      // <<  
    SHR      // >>  
    AND_NOT  // &^  
  
    ADD_ASSIGN // +=  
    SUB_ASSIGN // -=  
    MUL_ASSIGN // *=  
    QUO_ASSIGN // /=  
    REM_ASSIGN // %=  
  
    AND_ASSIGN      // &=  
    OR_ASSIGN       // |=  
    XOR_ASSIGN      // ^=  
    SHL_ASSIGN      // <<=  
    SHR_ASSIGN      // >>=  
    AND_NOT_ASSIGN  // &^=
```

```
LAND // &&
LOR // ||
ARROW // <-
INC // ++
DEC // --

EQL // ==
LSS // <
GTR // >
ASSIGN // =
NOT // !

NEQ // !=
LEQ // <=
GEQ // >=
DEFINE // :=
ELLIPSIS // ...

LPAREN // (
LBRACK // [
LBRACE // {
COMMA // ,
PERIOD // .

RPAREN // )
RBRACK // ]
RBRACE // }
SEMICOLON // ;
COLON

// Keywords
BREAK
CASE
CHAN
CONST
CONTINUE

DEFAULT
DEFER
ELSE
FALLTHROUGH
FOR

FUNC
GO
GOTO
IF
IMPORT
```

```
INTERFACE
MAP
PACKAGE
RANGE
RETURN

SELECT
STRUCT
SWITCH
TYPE
VAR
)
```

The list of tokens.

func [Lookup](#)

```
func Lookup(ident string) Token
```

Lookup maps an identifier to its keyword token or IDENT (if not a keyword).

func (Token) [IsKeyword](#)

```
func (tok Token) IsKeyword() bool
```

IsKeyword returns true for tokens corresponding to keywords; it returns false otherwise.

func (Token) [IsLiteral](#)

```
func (tok Token) IsLiteral() bool
```

IsLiteral returns true for tokens corresponding to identifiers and basic type literals; it returns false otherwise.

func (Token) [IsOperator](#)

```
func (tok Token) IsOperator() bool
```

IsOperator returns true for tokens corresponding to operators and delimiters; it returns false otherwise.

func (Token) [Precedence](#)

```
func (op Token) Precedence() int
```

Precedence returns the operator precedence of the binary operator op. If op is not a binary operator, the result is LowestPrecedence.

func (Token) [String](#)

```
func (tok Token) String() string
```

String returns the string corresponding to the token tok. For operators, delimiters, and keywords the string is the actual token character sequence (e.g., for the token ADD, the string is "+"). For all other tokens the string corresponds to the token constant name (e.g. for the token IDENT, the string is "IDENT").

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package hash

```
import "hash"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package hash provides interfaces for hash functions.

Index

[type Hash](#)

[type Hash32](#)

[type Hash64](#)

Package files

[hash.go](#)

type [Hash](#)

```
type Hash interface {
    // Write adds more data to the running hash.
    // It never returns an error.
    io.Writer

    // Sum appends the current hash to b and returns the resulting s
    // It does not change the underlying hash state.
    Sum(b []byte) []byte

    // Reset resets the hash to one with zero bytes written.
    Reset()

    // Size returns the number of bytes Sum will return.
    Size() int

    // BlockSize returns the hash's underlying block size.
    // The Write method must be able to accept any amount
    // of data, but it may operate more efficiently if all writes
    // are a multiple of the block size.
    BlockSize() int
}
```

Hash is the common interface implemented by all hash functions.

type [Hash32](#)

```
type Hash32 interface {  
    Hash  
    Sum32() uint32  
}
```

Hash32 is the common interface implemented by all 32-bit hash functions.

type [Hash64](#)

```
type Hash64 interface {  
    Hash  
    Sum64() uint64  
}
```

Hash64 is the common interface implemented by all 64-bit hash functions.

Subdirectories

Name	Synopsis
adler32	Package adler32 implements the Adler-32 checksum.
crc32	Package crc32 implements the 32-bit cyclic redundancy check, or CRC-32, checksum.
crc64	Package crc64 implements the 64-bit cyclic redundancy check, or CRC-64, checksum.
fnv	Package fnv implements FNV-1 and FNV-1a, non-cryptographic hash functions created by Glenn Fowler, Landon Curt Noll, and Phong Vo.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package `adler32`

```
import "hash/adler32"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `adler32` implements the Adler-32 checksum. Defined in RFC 1950:

Adler-32 is composed of two sums accumulated per byte: `s1` is the sum of all bytes, `s2` is the sum of all `s1` values. Both sums are done modulo 65521. `s1` is initialized to 1, `s2` to zero. The Adler-32 checksum is stored as $s2 * 65536 + s1$ in most-significant-byte first (network) order.

Index

Constants

[func Checksum\(data \[\]byte\) uint32](#)

[func New\(\) hash.Hash32](#)

Package files

[adler32.go](#)

Constants

```
const Size = 4
```

The size of an Adler-32 checksum in bytes.

func [Checksum](#)

```
func Checksum(data []byte) uint32
```

Checksum returns the Adler-32 checksum of data.

func [New](#)

```
func New() hash.Hash32
```

New returns a new hash.Hash32 computing the Adler-32 checksum.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package crc32

```
import "hash/crc32"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `crc32` implements the 32-bit cyclic redundancy check, or CRC-32, checksum. See http://en.wikipedia.org/wiki/Cyclic_redundancy_check for information.

Index

[Constants](#)

[Variables](#)

[func Checksum\(data \[\]byte, tab *Table\) uint32](#)

[func ChecksumIEEE\(data \[\]byte\) uint32](#)

[func New\(tab *Table\) hash.Hash32](#)

[func NewIEEE\(\) hash.Hash32](#)

[func Update\(crc uint32, tab *Table, p \[\]byte\) uint32](#)

[type Table](#)

[func MakeTable\(poly uint32\) *Table](#)

Package files

[crc32.go](#) [crc32_amd64.go](#)

Constants

```
const (  
    // Far and away the most common CRC-32 polynomial.  
    // Used by ethernet (IEEE 802.3), v.42, fddi, gzip, zip, png, mp  
    IEEE = 0xedb88320  
  
    // Castagnoli's polynomial, used in iSCSI.  
    // Has better error detection characteristics than IEEE.  
    // http://dx.doi.org/10.1109/26.231911  
    Castagnoli = 0x82f63b78  
  
    // Koopman's polynomial.  
    // Also has better error detection characteristics than IEEE.  
    // http://dx.doi.org/10.1109/DSN.2002.1028931  
    Koopman = 0xeb31d82e  
)
```

Predefined polynomials.

```
const Size = 4
```

The size of a CRC-32 checksum in bytes.

Variables

```
var IEEETable = makeTable(IEEE)
```

IEEETable is the table for the IEEE polynomial.

func [Checksum](#)

```
func Checksum(data []byte, tab *Table) uint32
```

Checksum returns the CRC-32 checksum of data using the polynomial represented by the Table.

func [ChecksumIEEE](#)

```
func ChecksumIEEE(data []byte) uint32
```

ChecksumIEEE returns the CRC-32 checksum of data using the IEEE polynomial.

func [New](#)

```
func New(tab *Table) hash.Hash32
```

New creates a new hash.Hash32 computing the CRC-32 checksum using the polynomial represented by the Table.

func [NewIEEE](#)

```
func NewIEEE() hash.Hash32
```

NewIEEE creates a new hash.Hash32 computing the CRC-32 checksum using the IEEE polynomial.

func [Update](#)

```
func Update(crc uint32, tab *Table, p []byte) uint32
```

Update returns the result of adding the bytes in p to the crc.

type [Table](#)

```
type Table [256]uint32
```

Table is a 256-word table representing the polynomial for efficient processing.

func [MakeTable](#)

```
func MakeTable(poly uint32) *Table
```

MakeTable returns the Table constructed from the specified polynomial.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package crc64

```
import "hash/crc64"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `crc64` implements the 64-bit cyclic redundancy check, or CRC-64, checksum. See http://en.wikipedia.org/wiki/Cyclic_redundancy_check for information.

Index

Constants

[func Checksum\(data \[\]byte, tab *Table\) uint64](#)

[func New\(tab *Table\) hash.Hash64](#)

[func Update\(crc uint64, tab *Table, p \[\]byte\) uint64](#)

[type Table](#)

[func MakeTable\(poly uint64\) *Table](#)

Package files

[crc64.go](#)

Constants

```
const (  
    // The ISO polynomial, defined in ISO 3309 and used in HDLC.  
    ISO = 0xD800000000000000  
  
    // The ECMA polynomial, defined in ECMA 182.  
    ECMA = 0xC96C5795D7870F42  
)
```

Predefined polynomials.

```
const Size = 8
```

The size of a CRC-64 checksum in bytes.

func [Checksum](#)

```
func Checksum(data []byte, tab *Table) uint64
```

Checksum returns the CRC-64 checksum of data using the polynomial represented by the Table.

func [New](#)

```
func New(tab *Table) hash.Hash64
```

New creates a new hash.Hash64 computing the CRC-64 checksum using the polynomial represented by the Table.

func [Update](#)

```
func Update(crc uint64, tab *Table, p []byte) uint64
```

Update returns the result of adding the bytes in p to the crc.

type [Table](#)

```
type Table [256]uint64
```

Table is a 256-word table representing the polynomial for efficient processing.

func [MakeTable](#)

```
func MakeTable(poly uint64) *Table
```

MakeTable returns the Table constructed from the specified polynomial.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package fnv

```
import "hash/fnv"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `fnv` implements FNV-1 and FNV-1a, non-cryptographic hash functions created by Glenn Fowler, Landon Curt Noll, and Phong Vo. See <http://isthe.com/chongo/tech/comp/fnv/>.

Index

[func New32\(\) hash.Hash32](#)
[func New32a\(\) hash.Hash32](#)
[func New64\(\) hash.Hash64](#)
[func New64a\(\) hash.Hash64](#)

Package files

[fnv.go](#)

func [New32](#)

```
func New32() hash.Hash32
```

New32 returns a new 32-bit FNV-1 hash.Hash.

func [New32a](#)

```
func New32a() hash.Hash32
```

New32a returns a new 32-bit FNV-1a hash.Hash.

func [New64](#)

```
func New64() hash.Hash64
```

New64 returns a new 64-bit FNV-1 hash.Hash.

func [New64a](#)

```
func New64a() hash.Hash64
```

New64a returns a new 64-bit FNV-1a hash.Hash.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package html

```
import "html"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package `html` provides functions for escaping and unescaping HTML text.

Index

[func EscapeString\(s string\) string](#)

[func UnescapeString\(s string\) string](#)

Package files

[entity.go](#) [escape.go](#)

func EscapeString

```
func EscapeString(s string) string
```

EscapeString escapes special characters like "<" to become "<". It escapes only five such characters: <, >, &, ' and ". UnescapeString(EscapeString(s)) == s always holds, but the converse isn't always true.

func UnescapeString

```
func UnescapeString(s string) string
```

UnescapeString unescapes entities like "<" to become "<". It unescapes a larger range of entities than EscapeString escapes. For example, "á" unescapes to "", as does "á" and "&xE1;".

UnescapeString(EscapeString(s)) == s always holds, but the converse isn't always true.

Subdirectories

Name	Synopsis
template	Package template (html/template) implements data-driven templates for generating HTML output safe against code injection.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package template

```
import "html/template"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `html/template` implements data-driven templates for generating HTML output safe against code injection. It provides the same interface as package `text/template` and should be used instead of `text/template` whenever the output is HTML.

The documentation here focuses on the security features of the package. For information about how to program the templates themselves, see the documentation for `text/template`.

Introduction

This package wraps package `text/template` so you can share its template API to parse and execute HTML templates safely.

```
tmpl, err := template.New("name").Parse(...)
// Error checking elided
err = tmpl.Execute(out, data)
```

If successful, `tmpl` will now be injection-safe. Otherwise, `err` is an error defined in the docs for `ErrorCode`.

HTML templates treat data values as plain text which should be encoded so they can be safely embedded in an HTML document. The escaping is contextual, so actions can appear within JavaScript, CSS, and URI contexts.

The security model used by this package assumes that template authors are trusted, while `Execute`'s data parameter is not. More details are provided below.

Example

```
import "text/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}`)
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwne
```

produces

```
Hello, <script>alert('you have been pwned')</script>!
```

but the contextual autoescaping in html/template

```
import "html/template"
...
t, err := template.New("foo").Parse(`{{define "T"}}Hello, {{.}}!{{end}}
err = t.ExecuteTemplate(out, "T", "<script>alert('you have been pwne
```

produces safe, escaped HTML output

```
Hello, &lt;script&gt;alert(&#39;you have been pwned&#39;)&lt;/script
```

Contexts

This package understands HTML, CSS, JavaScript, and URIs. It adds sanitizing functions to each simple action pipeline, so given the excerpt

```
<a href="/search?q={{.}}">{{.}}</a>
```

At parse time each `{{.}}` is overwritten to add escaping functions as necessary. In this case it becomes

```
<a href="/search?q={{. | urlquery}}">{{. | html}}</a>
```

Errors

See the documentation of `ErrorCode` for details.

A fuller picture

The rest of this package comment may be skipped on first reading; it includes details necessary to understand escaping contexts and error messages. Most users will not need to understand these details.

Contexts

Assuming `{{.}}` is ``O'Reilly: How are <i>you</i>?``, the table below shows how `{{.}}` appears when used in the context to the left.

Context	<code>{{.}}</code> After
<code>{{.}}</code>	<code>O'Reilly: How are &lt;i&gt;you&lt;/i>?</code>
<code></code>	<code>O&#39;Reilly: How are you?</code>

<code></code>	<code>O&#39;Reilly: How are %3ci%3eyou%3c</code>
<code></code>	<code>O&#39;Reilly%3a%20How%20are%3ci%3e.</code>
<code></code>	<code>0\x27Reilly: How are \x3ci\x3eyou..</code>
<code></code>	<code>"0\x27Reilly: How are \x3ci\x3eyou.</code>
<code></code>	<code>0\x27Reilly: How are \x3ci\x3eyou..</code>

If used in an unsafe context, then the value might be filtered out:

Context	{{.}} After
<code></code>	<code>#ZgotmplZ</code>

since "O'Reilly:" is not an allowed protocol like "http:".

If {{.}} is the innocuous word, `left`, then it can appear more widely,

Context	{{.}} After
<code>{{.}}</code>	<code>left</code>
<code></code>	<code>left</code>
<code></code>	<code>left</code>
<code></code>	<code>left</code>
<code></code>	<code>left</code>
<code></code>	<code>left</code>
<code></code>	<code>left</code>
<code></code>	<code>left</code>
<code></code>	<code>left</code>
<code><style>p.{{.}} {color:red}</style></code>	<code>left</code>

Non-string values can be used in JavaScript contexts. If {{.}} is

```
[]struct{A,B string}{ "foo", "bar" }
```

in the escaped template

```
<script>var pair = {{.}};</script>
```

then the template output is

```
<script>var pair = {"A": "foo", "B": "bar"};</script>
```

See package json to understand how non-string content is marshalled for embedding in JavaScript contexts.

Typed Strings

By default, this package assumes that all pipelines produce a plain text string. It

adds escaping pipeline stages necessary to correctly and safely embed that plain text string in the appropriate context.

When a data value is not plain text, you can make sure it is not over-escaped by marking it with its type.

Types HTML, JS, URL, and others from content.go can carry safe content that is exempted from escaping.

The template

```
Hello, {{.}}!
```

can be invoked with

```
tmpl.Execute(out, HTML(`World`))
```

to produce

```
Hello, World!
```

instead of the

```
Hello, &lt;b&gt;World&lt;b&gt;!
```

that would have been produced if `{{.}}` was a regular string.

Security Model

http://js-quasis-libraries-and-repl.googlecode.com/svn/trunk/safetemplate.html#problem_definition defines "safe" as used by this package.

This package assumes that template authors are trusted, that `Execute`'s data parameter is not, and seeks to preserve the properties below in the face of untrusted data:

Structure Preservation Property: "... when a template author writes an HTML tag in a safe templating language, the browser will interpret the corresponding portion of the output as a tag regardless of the values of untrusted data, and similarly for other structures such as attribute boundaries and JS and CSS string

boundaries."

Code Effect Property: "... only code specified by the template author should run as a result of injecting the template output into a page and all code specified by the template author should run as a result of the same."

Least Surprise Property: "A developer (or code reviewer) familiar with HTML, CSS, and JavaScript, who knows that contextual autoescaping happens should be able to look at a `{{.}}` and correctly infer what sanitization happens."

Index

[func HTMLEscape\(w io.Writer, b \[\]byte\)](#)
[func HTMLEscapeString\(s string\) string](#)
[func HTMLEscaper\(args ...interface{}\) string](#)
[func JSEscape\(w io.Writer, b \[\]byte\)](#)
[func JSEscapeString\(s string\) string](#)
[func JSEscaper\(args ...interface{}\) string](#)
[func URLQueryEscaper\(args ...interface{}\) string](#)
[type CSS](#)
[type Error](#)
 [func \(e *Error\) Error\(\) string](#)
[type ErrorCode](#)
[type FuncMap](#)
[type HTML](#)
[type HTMLAttr](#)
[type JS](#)
[type JSStr](#)
[type Template](#)
 [func Must\(t *Template, err error\) *Template](#)
 [func New\(name string\) *Template](#)
 [func ParseFiles\(filenamees ...string\) \(*Template, error\)](#)
 [func ParseGlob\(pattern string\) \(*Template, error\)](#)
 [func \(t *Template\) AddParseTree\(name string, tree *parse.Tree\)](#)
[\(*Template, error\)](#)
 [func \(t *Template\) Clone\(\) \(*Template, error\)](#)
 [func \(t *Template\) Delims\(left, right string\) *Template](#)
 [func \(t *Template\) Execute\(wr io.Writer, data interface{}\) \(err error\)](#)
 [func \(t *Template\) ExecuteTemplate\(wr io.Writer, name string, data](#)
[interface{}\) error](#)
 [func \(t *Template\) Funcs\(funcMap FuncMap\) *Template](#)
 [func \(t *Template\) Lookup\(name string\) *Template](#)
 [func \(t *Template\) Name\(\) string](#)
 [func \(t *Template\) New\(name string\) *Template](#)
 [func \(t *Template\) Parse\(src string\) \(*Template, error\)](#)
 [func \(t *Template\) ParseFiles\(filenamees ...string\) \(*Template, error\)](#)
 [func \(t *Template\) ParseGlob\(pattern string\) \(*Template, error\)](#)

[func \(t *Template\) Templates\(\) \[\]*Template](#)
[type URL](#)

Package files

[attr.go](#) [content.go](#) [context.go](#) [css.go](#) [doc.go](#) [error.go](#) [escape.go](#) [html.go](#) [js.go](#) [template.go](#) [transition.go](#) [url.go](#)

func [HTMLEscape](#)

```
func HTMLEscape(w io.Writer, b []byte)
```

HTMLEscape writes to w the escaped HTML equivalent of the plain text data b.

func [HTMLEscapeString](#)

```
func HTMLEscapeString(s string) string
```

HTMLEscapeString returns the escaped HTML equivalent of the plain text data s.

func [HTMLEscaper](#)

```
func HTMLEscaper(args ...interface{}) string
```

HTMLEscaper returns the escaped HTML equivalent of the textual representation of its arguments.

func [JSEscape](#)

```
func JSEscape(w io.Writer, b []byte)
```

JSEscape writes to w the escaped JavaScript equivalent of the plain text data b.

func JSEscapeString

```
func JSEscapeString(s string) string
```

JSEscapeString returns the escaped JavaScript equivalent of the plain text data s.

func [JSEscaper](#)

```
func JSEscaper(args ...interface{}) string
```

JSEscaper returns the escaped JavaScript equivalent of the textual representation of its arguments.

func [URLQueryEscaper](#)

```
func URLQueryEscaper(args ...interface{}) string
```

URLQueryEscaper returns the escaped value of the textual representation of its arguments in a form suitable for embedding in a URL query.

type [CSS](#)

type CSS string

CSS encapsulates known safe content that matches any of:

1. The CSS3 stylesheet production, such as ``p { color: purple }``.
2. The CSS3 rule production, such as ``a[href=~"https:"].foo#bar``.
3. CSS3 declaration productions, such as ``color: red; margin: 2px``.
4. The CSS3 value production, such as ``rgba(0, 0, 255, 127)``.

See <http://www.w3.org/TR/css3-syntax/#style>

type [Error](#)

```
type Error struct {  
    // ErrorCode describes the kind of error.  
    ErrorCode ErrorCode  
    // Name is the name of the template in which the error was encou  
    Name string  
    // Line is the line number of the error in the template source o  
    Line int  
    // Description is a human-readable description of the problem.  
    Description string  
}
```

Error describes a problem encountered during template Escaping.

func (***Error**) [Error](#)

```
func (e *Error) Error() string
```

type [ErrorCode](#)

```
type ErrorCode int
```

ErrorCode is a code for a kind of error.

```
const (
    // OK indicates the lack of an error.
    OK ErrorCode = iota

    // ErrAmbigContext: "... appears in an ambiguous URL context"
    // Example:
    //   <a href="
    //       {{if .C}}
    //         /path/
    //       {{else}}
    //         /search?q=
    //       {{end}}
    //       {{.X}}
    //   ">
    // Discussion:
    //   {{.X}} is in an ambiguous URL context since, depending on {
    //   it may be either a URL suffix or a query parameter.
    //   Moving {{.X}} into the condition removes the ambiguity:
    //   <a href="{{if .C}}/path/{{.X}}{{else}}/search?q={{.X}}">
    ErrAmbigContext

    // ErrBadHTML: "expected space, attr name, or end of tag, but go
    //   "... in unquoted attr", "... in attribute name"
    // Example:
    //   <a href = /search?q=foo>
    //   <href=foo>
    //   <form na<e=...>
    //   <option selected<
    // Discussion:
    //   This is often due to a typo in an HTML element, but some ru
    //   are banned in tag names, attribute names, and unquoted attr
    //   values because they can tickle parser ambiguities.
    //   Quoting all attributes is the best policy.
    ErrBadHTML

    // ErrBranchEnd: "{{if}} branches end in different contexts"
    // Example:
    //   {{if .C}}<a href="{{end}}{{.X}}
    // Discussion:
    //   Package html/template statically examines each path through
    //   {{if}}, {{range}}, or {{with}} to escape any following pipe
```

```

// The example is ambiguous since {{.X}} might be an HTML text
// or a URL prefix in an HTML attribute. The context of {{.X}}
// used to figure out how to escape it, but that context depen
// the run-time value of {{.C}} which is not statically known.
//
// The problem is usually something like missing quotes or ang
// brackets, or can be avoided by refactoring to put the two c
// into different branches of an if, range or with. If the pro
// is in a {{range}} over a collection that should never be er
// adding a dummy {{else}} can help.
ErrBranchEnd

// ErrEndContext: "... ends in a non-text context: ..."
// Examples:
// <div
// <div title="no close quote>
// <script>f()
// Discussion:
// Executed templates should produce a DocumentFragment of HTML
// Templates that end without closing tags will trigger this e
// Templates that should not be used in an HTML context or tha
// produce incomplete Fragments should not be executed directl
//
// {{define "main"}} <script>{{template "helper"}}</script> {{
// {{define "helper"}} document.write(' <div title=" ') {{end}}
//
// "helper" does not produce a valid document fragment, so sho
// not be Executed directly.
ErrEndContext

// ErrNoSuchTemplate: "no such template ..."
// Examples:
// {{define "main"}}<div {{template "attrs"}}>{{end}}
// {{define "attrs"}}href="{{.URL}}"{{end}}
// Discussion:
// Package html/template looks through template calls to compu
// context.
// Here the {{.URL}} in "attrs" must be treated as a URL when
// from "main", but you will get this error if "attrs" is not
// when "main" is parsed.
ErrNoSuchTemplate

// ErrOutputContext: "cannot compute output context for template
// Examples:
// {{define "t"}}{{if .T}}{{template "t" .T}}{{end}}{{.H}}",{{
// Discussion:
// A recursive template does not end in the same context in wh
// starts, and a reliable output context cannot be computed.
// Look for typos in the named template.

```

```

// If the template should not be called in the named start con
// look for calls to that template in unexpected contexts.
// Maybe refactor recursive templates to not be recursive.
ErrOutputContext

// ErrPartialCharset: "unfinished JS regexp charset in ..."
// Example:
//   <script>var pattern = /foo[{{.Chars}}]/</script>
// Discussion:
//   Package html/template does not support interpolation into r
//   expression literal character sets.
ErrPartialCharset

// ErrPartialEscape: "unfinished escape sequence in ..."
// Example:
//   <script>alert("\{{.X}}")</script>
// Discussion:
//   Package html/template does not support actions following a
//   backslash.
//   This is usually an error and there are better solutions; fo
//   example
//   <script>alert("{{.X}}")</script>
//   should work, and if {{.X}} is a partial escape sequence suc
//   "xA0", mark the whole sequence as safe content: JSStr(`xA0
ErrPartialEscape

// ErrRangeLoopReentry: "on range loop re-entry: ..."
// Example:
//   <script>var x = [{{range .}}'{{.}},{{end}}]</script>
// Discussion:
//   If an iteration through a range would cause it to end in a
//   different context than an earlier pass, there is no single
//   In the example, there is missing a quote, so it is not clea
//   whether {{.}} is meant to be inside a JS string or in a JS
//   context. The second iteration would produce something like
//
//   <script>var x = ['firstValue,'secondValue]</script>
ErrRangeLoopReentry

// ErrSlashAmbig: '/' could start a division or regexp.
// Example:
//   <script>
//     {{if .C}}var x = 1{{end}}
//     /-{{.N}}/i.test(x) ? doThis : doThat();
//   </script>
// Discussion:
//   The example above could produce `var x = 1/-2/i.test(s)...`
//   in which the first '/' is a mathematical division operator
//   could produce `/-2/i.test(s)` in which the first '/' starts

```

```
//  regexp literal.
//  Look for missing semicolons inside branches, and maybe add
//  parentheses to make it clear which interpretation you intend
ErrSlashAmbig
)
```

We define codes for each error that manifests while escaping templates, but escaped templates may also fail at runtime.

Output: "ZgotmplZ" Example:

```

where {{.X}} evaluates to `javascript:...`
```

Discussion:

"ZgotmplZ" is a special value that indicates that unsafe content reached CSS or URL context at runtime. The output of the example will be

```

If the data comes from a trusted source, use content types to exempt
from filtering: URL(`javascript:...`).
```

type [FuncMap](#)

```
type FuncMap map[string]interface{}
```

FuncMap is the type of the map defining the mapping from names to functions. Each function must have either a single return value, or two return values of which the second has type error. In that case, if the second (error) argument evaluates to non-nil during execution, execution terminates and Execute returns that error. FuncMap has the same base type as `template.FuncMap`, copied here so clients need not import "text/template".

type [HTML](#)

type HTML string

HTML encapsulates a known safe HTML document fragment. It should not be used for HTML from a third-party, or HTML with unclosed tags or comments. The outputs of a sound HTML sanitizer and a template escaped by this package are fine for use with HTML.

type [HTMLAttr](#)

```
type HTMLAttr string
```

HTMLAttr encapsulates an HTML attribute from a trusted source, for example, `dir="ltr"`.

type [JS](#)

type JS string

JS encapsulates a known safe EcmaScript5 Expression, for example, `(x + y * z())`. Template authors are responsible for ensuring that typed expressions do not break the intended precedence and that there is no statement/expression ambiguity as when passing an expression like `{ foo: bar() }\n['foo']()`, which is both a valid Expression and a valid Program with a very different meaning.

type [JSStr](#)

```
type JSStr string
```

JSStr encapsulates a sequence of characters meant to be embedded between quotes in a JavaScript expression. The string must match a series of `StringCharacter`s:

```
StringCharacter :: SourceCharacter but not `` or LineTerminator  
                | EscapeSequence
```

Note that `LineContinuations` are not allowed. `JSStr("foo\nbar")` is fine, but `JSStr("foo\\nbar")` is not.

type [Template](#)

```
type Template struct {  
    // contains filtered or unexported fields  
}
```

Template is a specialized template.Template that produces a safe HTML document fragment.

func [Must](#)

```
func Must(t *Template, err error) *Template
```

Must panics if err is non-nil in the same way as template.Must.

func [New](#)

```
func New(name string) *Template
```

New allocates a new HTML template with the given name.

func [ParseFiles](#)

```
func ParseFiles(filenamees ...string) (*Template, error)
```

ParseFiles creates a new Template and parses the template definitions from the named files. The returned template's name will have the (base) name and (parsed) contents of the first file. There must be at least one file. If an error occurs, parsing stops and the returned *Template is nil.

func [ParseGlob](#)

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob creates a new Template and parses the template definitions from the files identified by the pattern, which must match at least one file. The returned template will have the (base) name and (parsed) contents of the first file matched by the pattern. ParseGlob is equivalent to calling ParseFiles with the list of files matched by the pattern.

func (*Template) [AddParseTree](#)

```
func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Tem
```

AddParseTree creates a new template with the name and parse tree and associates it with t.

It returns an error if t has already been executed.

func (*Template) [Clone](#)

```
func (t *Template) Clone() (*Template, error)
```

Clone returns a duplicate of the template, including all associated templates. The actual representation is not copied, but the name space of associated templates is, so further calls to Parse in the copy will add templates to the copy but not to the original. Clone can be used to prepare common templates and use them with variant definitions for other templates by adding the variants after the clone is made.

It returns an error if t has already been executed.

func (*Template) [Delims](#)

```
func (t *Template) Delims(left, right string) *Template
```

Delims sets the action delimiters to the specified strings, to be used in subsequent calls to Parse, ParseFiles, or ParseGlob. Nested template definitions will inherit the settings. An empty delimiter stands for the corresponding default: {{ or }}. The return value is the template, so calls can be chained.

func (*Template) [Execute](#)

```
func (t *Template) Execute(wr io.Writer, data interface{}) (err error)
```

Execute applies a parsed template to the specified data object, writing the output to wr.

func (*Template) [ExecuteTemplate](#)

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data i
```

ExecuteTemplate applies the template associated with t that has the given name to the specified data object and writes the output to wr.

func (*Template) [Funcs](#)

```
func (t *Template) Funcs(funcMap FuncMap) *Template
```

Funcs adds the elements of the argument map to the template's function map. It panics if a value in the map is not a function with appropriate return type. However, it is legal to overwrite elements of the map. The return value is the template, so calls can be chained.

func (*Template) [Lookup](#)

```
func (t *Template) Lookup(name string) *Template
```

Lookup returns the template with the given name that is associated with t, or nil if there is no such template.

func (*Template) [Name](#)

```
func (t *Template) Name() string
```

Name returns the name of the template.

func (*Template) [New](#)

```
func (t *Template) New(name string) *Template
```

New allocates a new HTML template associated with the given one and with the same delimiters. The association, which is transitive, allows one template to invoke another with a {{template}} action.

func (*Template) [Parse](#)

```
func (t *Template) Parse(src string) (*Template, error)
```

Parse parses a string into a template. Nested template definitions will be

associated with the top-level template `t`. `Parse` may be called multiple times to parse definitions of templates to associate with `t`. It is an error if a resulting template is non-empty (contains content other than template definitions) and would replace a non-empty template with the same name. (In multiple calls to `Parse` with the same receiver template, only one call can contain text other than space, comments, and template definitions.)

func (*Template) [ParseFiles](#)

```
func (t *Template) ParseFiles(filenamees ...string) (*Template, error)
```

`ParseFiles` parses the named files and associates the resulting templates with `t`. If an error occurs, parsing stops and the returned template is `nil`; otherwise it is `t`. There must be at least one file.

func (*Template) [ParseGlob](#)

```
func (t *Template) ParseGlob(pattern string) (*Template, error)
```

`ParseGlob` parses the template definitions in the files identified by the pattern and associates the resulting templates with `t`. The pattern is processed by `filepath.Glob` and must match at least one file. `ParseGlob` is equivalent to calling `t.ParseFiles` with the list of files matched by the pattern.

func (*Template) [Templates](#)

```
func (t *Template) Templates() []*Template
```

`Templates` returns a slice of the templates associated with `t`, including `t` itself.

type [URL](#)

type URL string

URL encapsulates a known safe URL as defined in RFC 3896. A URL like ``javascript:checkThatFormNotEditedBeforeLeavingPage()`` from a trusted source should go in the page, but by default dynamic ``javascript:`` URLs are filtered out since they are a frequently exploited injection vector.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package image

```
import "image"
```

[Overview](#)

[Index](#)

[Examples](#)

[Subdirectories](#)

Overview ?

Overview ?

Package image implements a basic 2-D image library.

The fundamental interface is called Image. An Image contains colors, which are described in the image/color package.

Values of the Image interface are created either by calling functions such as NewRGBA and NewPaletted, or by calling Decode on an io.Reader containing image data in a format such as GIF, JPEG or PNG. Decoding any particular image format requires the prior registration of a decoder function. Registration is typically automatic as a side effect of initializing that format's package so that, to decode a PNG image, it suffices to have

```
import _ "image/png"
```

in a program's main package. The _ means to import a package purely for its initialization side effects.

See "The Go image package" for more details:

http://golang.org/doc/articles/image_package.html

? Example

? Example

Code:

```
// Open the file.
file, err := os.Open("testdata/video-001.jpeg")
if err != nil {
    log.Fatal(err)
}
defer file.Close()

// Decode the image.
m, _, err := image.Decode(file)
if err != nil {
    log.Fatal(err)
}
bounds := m.Bounds()
```

```

// Calculate a 16-bin histogram for m's red, green, blue and alpha c
//
// An image's bounds do not necessarily start at (0, 0), so the two
// at bounds.Min.Y and bounds.Min.X. Looping over Y first and X seco
// likely to result in better memory access patterns than X first an
var histogram [16][4]int
for y := bounds.Min.Y; y < bounds.Max.Y; y++ {
    for x := bounds.Min.X; x < bounds.Max.X; x++ {
        r, g, b, a := m.At(x, y).RGBA()
        // A color's RGBA method returns values in the range [0, 655
        // Shifting by 12 reduces this to the range [0, 15].
        histogram[r>>12][0]++
        histogram[g>>12][1]++
        histogram[b>>12][2]++
        histogram[a>>12][3]++
    }
}

// Print the results.
fmt.Printf("%-14s %6s %6s %6s %6s\n", "bin", "red", "green", "blue",
for i, x := range histogram {
    fmt.Printf("0x%04x-0x%04x: %6d %6d %6d %6d\n", i<<12, (i+1)<<12-
}

```

Output:

bin	red	green	blue	alpha
0x0000-0x0fff:	471	819	7596	0
0x1000-0x1fff:	576	2892	726	0
0x2000-0x2fff:	1038	2330	943	0
0x3000-0x3fff:	883	2321	1014	0
0x4000-0x4fff:	501	1295	525	0
0x5000-0x5fff:	302	962	242	0
0x6000-0x6fff:	219	358	150	0
0x7000-0x7fff:	352	281	192	0
0x8000-0x8fff:	3688	216	246	0
0x9000-0x9fff:	2277	237	283	0
0xa000-0xafff:	971	254	357	0
0xb000-0xbfff:	317	306	429	0
0xc000-0xcfff:	203	402	401	0
0xd000-0xdfff:	256	394	241	0
0xe000-0xefff:	378	343	173	0
0xf000-0xffff:	3018	2040	1932	15450

Index

Variables

func RegisterFormat(name, magic string, decode func(io.Reader) (Image, error), decodeConfig func(io.Reader) (Config, error))

type Alpha

func NewAlpha(r Rectangle) *Alpha

func (p *Alpha) At(x, y int) color.Color

func (p *Alpha) Bounds() Rectangle

func (p *Alpha) ColorModel() color.Model

func (p *Alpha) Opaque() bool

func (p *Alpha) PixOffset(x, y int) int

func (p *Alpha) Set(x, y int, c color.Color)

func (p *Alpha) SetAlpha(x, y int, c color.Alpha)

func (p *Alpha) SubImage(r Rectangle) Image

type Alpha16

func NewAlpha16(r Rectangle) *Alpha16

func (p *Alpha16) At(x, y int) color.Color

func (p *Alpha16) Bounds() Rectangle

func (p *Alpha16) ColorModel() color.Model

func (p *Alpha16) Opaque() bool

func (p *Alpha16) PixOffset(x, y int) int

func (p *Alpha16) Set(x, y int, c color.Color)

func (p *Alpha16) SetAlpha16(x, y int, c color.Alpha16)

func (p *Alpha16) SubImage(r Rectangle) Image

type Config

func DecodeConfig(r io.Reader) (Config, string, error)

type Gray

func NewGray(r Rectangle) *Gray

func (p *Gray) At(x, y int) color.Color

func (p *Gray) Bounds() Rectangle

func (p *Gray) ColorModel() color.Model

func (p *Gray) Opaque() bool

func (p *Gray) PixOffset(x, y int) int

func (p *Gray) Set(x, y int, c color.Color)

func (p *Gray) SetGray(x, y int, c color.Gray)

func (p *Gray) SubImage(r Rectangle) Image

type Gray16

func NewGray16(r Rectangle) *Gray16

func (p *Gray16) At(x, y int) color.Color

func (p *Gray16) Bounds() Rectangle

func (p *Gray16) ColorModel() color.Model

func (p *Gray16) Opaque() bool

func (p *Gray16) PixOffset(x, y int) int

func (p *Gray16) Set(x, y int, c color.Color)

func (p *Gray16) SetGray16(x, y int, c color.Gray16)

func (p *Gray16) SubImage(r Rectangle) Image

type Image

func Decode(r io.Reader) (Image, string, error)

type NRGBA

func NewNRGBA(r Rectangle) *NRGBA

func (p *NRGBA) At(x, y int) color.Color

func (p *NRGBA) Bounds() Rectangle

func (p *NRGBA) ColorModel() color.Model

func (p *NRGBA) Opaque() bool

func (p *NRGBA) PixOffset(x, y int) int

func (p *NRGBA) Set(x, y int, c color.Color)

func (p *NRGBA) SetNRGBA(x, y int, c color.NRGBA)

func (p *NRGBA) SubImage(r Rectangle) Image

type NRGBA64

func NewNRGBA64(r Rectangle) *NRGBA64

func (p *NRGBA64) At(x, y int) color.Color

func (p *NRGBA64) Bounds() Rectangle

func (p *NRGBA64) ColorModel() color.Model

func (p *NRGBA64) Opaque() bool

func (p *NRGBA64) PixOffset(x, y int) int

func (p *NRGBA64) Set(x, y int, c color.Color)

func (p *NRGBA64) SetNRGBA64(x, y int, c color.NRGBA64)

func (p *NRGBA64) SubImage(r Rectangle) Image

type Paletted

func NewPaletted(r Rectangle, p color.Palette) *Paletted

func (p *Paletted) At(x, y int) color.Color

func (p *Paletted) Bounds() Rectangle

func (p *Paletted) ColorIndexAt(x, y int) uint8

func (p *Paletted) ColorModel() color.Model

func (p *Paletted) Opaque() bool

[func \(p *Paletted\) PixOffset\(x, y int\) int](#)
[func \(p *Paletted\) Set\(x, y int, c color.Color\)](#)
[func \(p *Paletted\) SetColorIndex\(x, y int, index uint8\)](#)
[func \(p *Paletted\) SubImage\(r Rectangle\) Image](#)

[type PalettedImage](#)

[type Point](#)

[func Pt\(X, Y int\) Point](#)
[func \(p Point\) Add\(q Point\) Point](#)
[func \(p Point\) Div\(k int\) Point](#)
[func \(p Point\) Eq\(q Point\) bool](#)
[func \(p Point\) In\(r Rectangle\) bool](#)
[func \(p Point\) Mod\(r Rectangle\) Point](#)
[func \(p Point\) Mul\(k int\) Point](#)
[func \(p Point\) String\(\) string](#)
[func \(p Point\) Sub\(q Point\) Point](#)

[type RGBA](#)

[func NewRGBA\(r Rectangle\) *RGBA](#)
[func \(p *RGBA\) At\(x, y int\) color.Color](#)
[func \(p *RGBA\) Bounds\(\) Rectangle](#)
[func \(p *RGBA\) ColorModel\(\) color.Model](#)
[func \(p *RGBA\) Opaque\(\) bool](#)
[func \(p *RGBA\) PixOffset\(x, y int\) int](#)
[func \(p *RGBA\) Set\(x, y int, c color.Color\)](#)
[func \(p *RGBA\) SetRGBA\(x, y int, c color.RGBA\)](#)
[func \(p *RGBA\) SubImage\(r Rectangle\) Image](#)

[type RGBA64](#)

[func NewRGBA64\(r Rectangle\) *RGBA64](#)
[func \(p *RGBA64\) At\(x, y int\) color.Color](#)
[func \(p *RGBA64\) Bounds\(\) Rectangle](#)
[func \(p *RGBA64\) ColorModel\(\) color.Model](#)
[func \(p *RGBA64\) Opaque\(\) bool](#)
[func \(p *RGBA64\) PixOffset\(x, y int\) int](#)
[func \(p *RGBA64\) Set\(x, y int, c color.Color\)](#)
[func \(p *RGBA64\) SetRGBA64\(x, y int, c color.RGBA64\)](#)
[func \(p *RGBA64\) SubImage\(r Rectangle\) Image](#)

[type Rectangle](#)

[func Rect\(x0, y0, x1, y1 int\) Rectangle](#)
[func \(r Rectangle\) Add\(p Point\) Rectangle](#)
[func \(r Rectangle\) Canon\(\) Rectangle](#)

[func \(r Rectangle\) Dx\(\) int](#)
[func \(r Rectangle\) Dy\(\) int](#)
[func \(r Rectangle\) Empty\(\) bool](#)
[func \(r Rectangle\) Eq\(s Rectangle\) bool](#)
[func \(r Rectangle\) In\(s Rectangle\) bool](#)
[func \(r Rectangle\) Inset\(n int\) Rectangle](#)
[func \(r Rectangle\) Intersect\(s Rectangle\) Rectangle](#)
[func \(r Rectangle\) Overlaps\(s Rectangle\) bool](#)
[func \(r Rectangle\) Size\(\) Point](#)
[func \(r Rectangle\) String\(\) string](#)
[func \(r Rectangle\) Sub\(p Point\) Rectangle](#)
[func \(r Rectangle\) Union\(s Rectangle\) Rectangle](#)
[type Uniform](#)
[func NewUniform\(c color.Color\) *Uniform](#)
[func \(c *Uniform\) At\(x, y int\) color.Color](#)
[func \(c *Uniform\) Bounds\(\) Rectangle](#)
[func \(c *Uniform\) ColorModel\(\) color.Model](#)
[func \(c *Uniform\) Convert\(color.Color\) color.Color](#)
[func \(c *Uniform\) Opaque\(\) bool](#)
[func \(c *Uniform\) RGBA\(\) \(r, g, b, a uint32\)](#)
[type YCbCr](#)
[func NewYCbCr\(r Rectangle, subsampleRatio YCbCrSubsampleRatio\) *YCbCr](#)
[func \(p *YCbCr\) At\(x, y int\) color.Color](#)
[func \(p *YCbCr\) Bounds\(\) Rectangle](#)
[func \(p *YCbCr\) COffset\(x, y int\) int](#)
[func \(p *YCbCr\) ColorModel\(\) color.Model](#)
[func \(p *YCbCr\) Opaque\(\) bool](#)
[func \(p *YCbCr\) SubImage\(r Rectangle\) Image](#)
[func \(p *YCbCr\) YOffset\(x, y int\) int](#)
[type YCbCrSubsampleRatio](#)
[func \(s YCbCrSubsampleRatio\) String\(\) string](#)

Examples

[Package](#)

Package files

[format.go](#) [geom.go](#) [image.go](#) [names.go](#) [ycbcr.go](#)

Variables

```
var (  
    // Black is an opaque black uniform image.  
    Black = NewUniform(color.Black)  
    // White is an opaque white uniform image.  
    White = NewUniform(color.White)  
    // Transparent is a fully transparent uniform image.  
    Transparent = NewUniform(color.Transparent)  
    // Opaque is a fully opaque uniform image.  
    Opaque = NewUniform(color.Opaque)  
)  
  
var ErrFormat = errors.New("image: unknown format")
```

ErrFormat indicates that decoding encountered an unknown format.

func RegisterFormat

```
func RegisterFormat(name, magic string, decode func(io.Reader) (Imag
```

RegisterFormat registers an image format for use by Decode. Name is the name of the format, like "jpeg" or "png". Magic is the magic prefix that identifies the format's encoding. The magic string can contain "?" wildcards that each match any one byte. Decode is the function that decodes the encoded image. DecodeConfig is the function that decodes just its configuration.

type [Alpha](#)

```
type Alpha struct {
    // Pix holds the image's pixels, as alpha values. The pixel at
    // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*1
    Pix []uint8
    // Stride is the Pix stride (in bytes) between vertically adjae
    Stride int
    // Rect is the image's bounds.
    Rect Rectangle
}
```

Alpha is an in-memory image whose `At` method returns `color.Alpha` values.

func [NewAlpha](#)

```
func NewAlpha(r Rectangle) *Alpha
```

`NewAlpha` returns a new `Alpha` with the given bounds.

func (*Alpha) [At](#)

```
func (p *Alpha) At(x, y int) color.Color
```

func (*Alpha) [Bounds](#)

```
func (p *Alpha) Bounds() Rectangle
```

func (*Alpha) [ColorModel](#)

```
func (p *Alpha) ColorModel() color.Model
```

func (*Alpha) [Opaque](#)

```
func (p *Alpha) Opaque() bool
```

`Opaque` scans the entire image and returns whether or not it is fully opaque.

func (*Alpha) [PixOffset](#)

```
func (p *Alpha) PixOffset(x, y int) int
```

PixOffset returns the index of the first element of Pix that corresponds to the pixel at (x, y).

func (*Alpha) [Set](#)

```
func (p *Alpha) Set(x, y int, c color.Color)
```

func (*Alpha) [SetAlpha](#)

```
func (p *Alpha) SetAlpha(x, y int, c color.Alpha)
```

func (*Alpha) [SubImage](#)

```
func (p *Alpha) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

type [Alpha16](#)

```
type Alpha16 struct {  
    // Pix holds the image's pixels, as alpha values in big-endian f  
    // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*2  
    Pix []uint8  
    // Stride is the Pix stride (in bytes) between vertically adja  
    Stride int  
    // Rect is the image's bounds.  
    Rect Rectangle  
}
```

Alpha16 is an in-memory image whose `At` method returns `color.Alpha64` values.

func [NewAlpha16](#)

```
func NewAlpha16(r Rectangle) *Alpha16
```

`NewAlpha16` returns a new `Alpha16` with the given bounds.

func (*Alpha16) [At](#)

```
func (p *Alpha16) At(x, y int) color.Color
```

func (*Alpha16) [Bounds](#)

```
func (p *Alpha16) Bounds() Rectangle
```

func (*Alpha16) [ColorModel](#)

```
func (p *Alpha16) ColorModel() color.Model
```

func (*Alpha16) [Opaque](#)

```
func (p *Alpha16) Opaque() bool
```

`Opaque` scans the entire image and returns whether or not it is fully opaque.

func (*Alpha16) [PixOffset](#)

```
func (p *Alpha16) PixOffset(x, y int) int
```

PixOffset returns the index of the first element of Pix that corresponds to the pixel at (x, y).

func (*Alpha16) [Set](#)

```
func (p *Alpha16) Set(x, y int, c color.Color)
```

func (*Alpha16) [SetAlpha16](#)

```
func (p *Alpha16) SetAlpha16(x, y int, c color.Alpha16)
```

func (*Alpha16) [SubImage](#)

```
func (p *Alpha16) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

type [Config](#)

```
type Config struct {  
    ColorModel    color.Model  
    Width, Height int  
}
```

Config holds an image's color model and dimensions.

func [DecodeConfig](#)

```
func DecodeConfig(r io.Reader) (Config, string, error)
```

DecodeConfig decodes the color model and dimensions of an image that has been encoded in a registered format. The string returned is the format name used during format registration. Format registration is typically done by the init method of the codec-specific package.

type [Gray](#)

```
type Gray struct {
    // Pix holds the image's pixels, as gray values. The pixel at
    // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*1
    Pix []uint8
    // Stride is the Pix stride (in bytes) between vertically adjae
    Stride int
    // Rect is the image's bounds.
    Rect Rectangle
}
```

Gray is an in-memory image whose `At` method returns `color.Gray` values.

func [NewGray](#)

```
func NewGray(r Rectangle) *Gray
```

`NewGray` returns a new `Gray` with the given bounds.

func (***Gray**) [At](#)

```
func (p *Gray) At(x, y int) color.Color
```

func (***Gray**) [Bounds](#)

```
func (p *Gray) Bounds() Rectangle
```

func (***Gray**) [ColorModel](#)

```
func (p *Gray) ColorModel() color.Model
```

func (***Gray**) [Opaque](#)

```
func (p *Gray) Opaque() bool
```

`Opaque` scans the entire image and returns whether or not it is fully opaque.

func (***Gray**) [PixOffset](#)

```
func (p *Gray) PixOffset(x, y int) int
```

PixOffset returns the index of the first element of Pix that corresponds to the pixel at (x, y).

func (*Gray) [Set](#)

```
func (p *Gray) Set(x, y int, c color.Color)
```

func (*Gray) [SetGray](#)

```
func (p *Gray) SetGray(x, y int, c color.Gray)
```

func (*Gray) [SubImage](#)

```
func (p *Gray) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

type [Gray16](#)

```
type Gray16 struct {  
    // Pix holds the image's pixels, as gray values in big-endian fo  
    // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*2  
    Pix []uint8  
    // Stride is the Pix stride (in bytes) between vertically adja  
    Stride int  
    // Rect is the image's bounds.  
    Rect Rectangle  
}
```

Gray16 is an in-memory image whose `At` method returns `color.Gray16` values.

func [NewGray16](#)

```
func NewGray16(r Rectangle) *Gray16
```

`NewGray16` returns a new `Gray16` with the given bounds.

func (*Gray16) [At](#)

```
func (p *Gray16) At(x, y int) color.Color
```

func (*Gray16) [Bounds](#)

```
func (p *Gray16) Bounds() Rectangle
```

func (*Gray16) [ColorModel](#)

```
func (p *Gray16) ColorModel() color.Model
```

func (*Gray16) [Opaque](#)

```
func (p *Gray16) Opaque() bool
```

`Opaque` scans the entire image and returns whether or not it is fully opaque.

func (*Gray16) [PixOffset](#)

```
func (p *Gray16) PixOffset(x, y int) int
```

PixOffset returns the index of the first element of Pix that corresponds to the pixel at (x, y).

func (*Gray16) [Set](#)

```
func (p *Gray16) Set(x, y int, c color.Color)
```

func (*Gray16) [SetGray16](#)

```
func (p *Gray16) SetGray16(x, y int, c color.Gray16)
```

func (*Gray16) [SubImage](#)

```
func (p *Gray16) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

type [Image](#)

```
type Image interface {
    // ColorModel returns the Image's color model.
    ColorModel() color.Model
    // Bounds returns the domain for which At can return non-zero co
    // The bounds do not necessarily contain the point (0, 0).
    Bounds() Rectangle
    // At returns the color of the pixel at (x, y).
    // At(Bounds().Min.X, Bounds().Min.Y) returns the upper-left pix
    // At(Bounds().Max.X-1, Bounds().Max.Y-1) returns the lower-right
    At(x, y int) color.Color
}
```

Image is a finite rectangular grid of color.Color values taken from a color model.

func [Decode](#)

```
func Decode(r io.Reader) (Image, string, error)
```

Decode decodes an image that has been encoded in a registered format. The string returned is the format name used during format registration. Format registration is typically done by the init method of the codec- specific package.

type [NRGBA](#)

```
type NRGBA struct {  
    // Pix holds the image's pixels, in R, G, B, A order. The pixel  
    // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*4  
    Pix []uint8  
    // Stride is the Pix stride (in bytes) between vertically adjae  
    Stride int  
    // Rect is the image's bounds.  
    Rect Rectangle  
}
```

NRGBA is an in-memory image whose `At` method returns `color.NRGBA` values.

func [NewNRGBA](#)

```
func NewNRGBA(r Rectangle) *NRGBA
```

`NewNRGBA` returns a new `NRGBA` with the given bounds.

func (*NRGBA) [At](#)

```
func (p *NRGBA) At(x, y int) color.Color
```

func (*NRGBA) [Bounds](#)

```
func (p *NRGBA) Bounds() Rectangle
```

func (*NRGBA) [ColorModel](#)

```
func (p *NRGBA) ColorModel() color.Model
```

func (*NRGBA) [Opaque](#)

```
func (p *NRGBA) Opaque() bool
```

`Opaque` scans the entire image and returns whether or not it is fully opaque.

func (*NRGBA) [PixOffset](#)

```
func (p *NRGBA) PixOffset(x, y int) int
```

PixOffset returns the index of the first element of Pix that corresponds to the pixel at (x, y).

func (*NRGBA) [Set](#)

```
func (p *NRGBA) Set(x, y int, c color.Color)
```

func (*NRGBA) [SetNRGBA](#)

```
func (p *NRGBA) SetNRGBA(x, y int, c color.NRGBA)
```

func (*NRGBA) [SubImage](#)

```
func (p *NRGBA) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

type [NRGBA64](#)

```
type NRGBA64 struct {  
    // Pix holds the image's pixels, in R, G, B, A order and big-end  
    // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*8  
    Pix []uint8  
    // Stride is the Pix stride (in bytes) between vertically adjace  
    Stride int  
    // Rect is the image's bounds.  
    Rect Rectangle  
}
```

NRGBA64 is an in-memory image whose `At` method returns `color.NRGBA64` values.

func [NewNRGBA64](#)

```
func NewNRGBA64(r Rectangle) *NRGBA64
```

`NewNRGBA64` returns a new `NRGBA64` with the given bounds.

func (***NRGBA64**) [At](#)

```
func (p *NRGBA64) At(x, y int) color.Color
```

func (***NRGBA64**) [Bounds](#)

```
func (p *NRGBA64) Bounds() Rectangle
```

func (***NRGBA64**) [ColorModel](#)

```
func (p *NRGBA64) ColorModel() color.Model
```

func (***NRGBA64**) [Opaque](#)

```
func (p *NRGBA64) Opaque() bool
```

`Opaque` scans the entire image and returns whether or not it is fully opaque.

func (***NRGBA64**) [PixOffset](#)

```
func (p *NRGBA64) PixOffset(x, y int) int
```

PixOffset returns the index of the first element of Pix that corresponds to the pixel at (x, y).

func (*NRGBA64) [Set](#)

```
func (p *NRGBA64) Set(x, y int, c color.Color)
```

func (*NRGBA64) [SetNRGBA64](#)

```
func (p *NRGBA64) SetNRGBA64(x, y int, c color.NRGBA64)
```

func (*NRGBA64) [SubImage](#)

```
func (p *NRGBA64) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

type [Paletted](#)

```
type Paletted struct {  
    // Pix holds the image's pixels, as palette indices. The pixel a  
    // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*1  
    Pix []uint8  
    // Stride is the Pix stride (in bytes) between vertically adja  
    Stride int  
    // Rect is the image's bounds.  
    Rect Rectangle  
    // Palette is the image's palette.  
    Palette color.Palette  
}
```

Paletted is an in-memory image of uint8 indices into a given palette.

func [NewPaletted](#)

```
func NewPaletted(r Rectangle, p color.Palette) *Paletted
```

NewPaletted returns a new Paletted with the given width, height and palette.

func (*Paletted) [At](#)

```
func (p *Paletted) At(x, y int) color.Color
```

func (*Paletted) [Bounds](#)

```
func (p *Paletted) Bounds() Rectangle
```

func (*Paletted) [ColorIndexAt](#)

```
func (p *Paletted) ColorIndexAt(x, y int) uint8
```

func (*Paletted) [ColorModel](#)

```
func (p *Paletted) ColorModel() color.Model
```

func (*Paletted) [Opaque](#)

```
func (p *Paletted) Opaque() bool
```

Opaque scans the entire image and returns whether or not it is fully opaque.

func (*Paletted) [PixOffset](#)

```
func (p *Paletted) PixOffset(x, y int) int
```

PixOffset returns the index of the first element of Pix that corresponds to the pixel at (x, y).

func (*Paletted) [Set](#)

```
func (p *Paletted) Set(x, y int, c color.Color)
```

func (*Paletted) [SetColorIndex](#)

```
func (p *Paletted) SetColorIndex(x, y int, index uint8)
```

func (*Paletted) [SubImage](#)

```
func (p *Paletted) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

type [PalettedImage](#)

```
type PalettedImage interface {  
    // ColorIndexAt returns the palette index of the pixel at (x, y)  
    ColorIndexAt(x, y int) uint8  
    Image  
}
```

PalettedImage is an image whose colors may come from a limited palette. If *m* is a PalettedImage and *m*.ColorModel() returns a PalettedColorModel *p*, then *m*.At(*x*, *y*) should be equivalent to *p*[*m*.ColorIndexAt(*x*, *y*)]. If *m*'s color model is not a PalettedColorModel, then ColorIndexAt's behavior is undefined.

type [Point](#)

```
type Point struct {  
    X, Y int  
}
```

A Point is an X, Y coordinate pair. The axes increase right and down.

```
var ZP Point
```

ZP is the zero Point.

func [Pt](#)

```
func Pt(X, Y int) Point
```

Pt is shorthand for Point{X, Y}.

func (Point) [Add](#)

```
func (p Point) Add(q Point) Point
```

Add returns the vector $p+q$.

func (Point) [Div](#)

```
func (p Point) Div(k int) Point
```

Div returns the vector p/k .

func (Point) [Eq](#)

```
func (p Point) Eq(q Point) bool
```

Eq returns whether p and q are equal.

func (Point) [In](#)

```
func (p Point) In(r Rectangle) bool
```

In returns whether p is in r.

func (Point) [Mod](#)

```
func (p Point) Mod(r Rectangle) Point
```

Mod returns the point q in r such that $p.X - q.X$ is a multiple of r's width and $p.Y - q.Y$ is a multiple of r's height.

func (Point) [Mul](#)

```
func (p Point) Mul(k int) Point
```

Mul returns the vector $p * k$.

func (Point) [String](#)

```
func (p Point) String() string
```

String returns a string representation of p like "(3,4)".

func (Point) [Sub](#)

```
func (p Point) Sub(q Point) Point
```

Sub returns the vector $p - q$.

type [RGBA](#)

```
type RGBA struct {  
    // Pix holds the image's pixels, in R, G, B, A order. The pixel  
    // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*4  
    Pix []uint8  
    // Stride is the Pix stride (in bytes) between vertically adjae  
    Stride int  
    // Rect is the image's bounds.  
    Rect Rectangle  
}
```

RGBA is an in-memory image whose `At` method returns `color.RGBA` values.

func [NewRGBA](#)

```
func NewRGBA(r Rectangle) *RGBA
```

`NewRGBA` returns a new `RGBA` with the given bounds.

func (***RGBA**) [At](#)

```
func (p *RGBA) At(x, y int) color.Color
```

func (***RGBA**) [Bounds](#)

```
func (p *RGBA) Bounds() Rectangle
```

func (***RGBA**) [ColorModel](#)

```
func (p *RGBA) ColorModel() color.Model
```

func (***RGBA**) [Opaque](#)

```
func (p *RGBA) Opaque() bool
```

`Opaque` scans the entire image and returns whether or not it is fully opaque.

func (***RGBA**) [PixOffset](#)

```
func (p *RGBA) PixOffset(x, y int) int
```

PixOffset returns the index of the first element of Pix that corresponds to the pixel at (x, y).

func (*RGBA) [Set](#)

```
func (p *RGBA) Set(x, y int, c color.Color)
```

func (*RGBA) [SetRGBA](#)

```
func (p *RGBA) SetRGBA(x, y int, c color.RGBA)
```

func (*RGBA) [SubImage](#)

```
func (p *RGBA) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

type [RGBA64](#)

```
type RGBA64 struct {  
    // Pix holds the image's pixels, in R, G, B, A order and big-end  
    // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-Rect.Min.X)*8  
    Pix []uint8  
    // Stride is the Pix stride (in bytes) between vertically adja  
    Stride int  
    // Rect is the image's bounds.  
    Rect Rectangle  
}
```

RGBA64 is an in-memory image whose `At` method returns `color.RGBA64` values.

func [NewRGBA64](#)

```
func NewRGBA64(r Rectangle) *RGBA64
```

`NewRGBA64` returns a new `RGBA64` with the given bounds.

func (*[RGBA64](#)) [At](#)

```
func (p *RGBA64) At(x, y int) color.Color
```

func (*[RGBA64](#)) [Bounds](#)

```
func (p *RGBA64) Bounds() Rectangle
```

func (*[RGBA64](#)) [ColorModel](#)

```
func (p *RGBA64) ColorModel() color.Model
```

func (*[RGBA64](#)) [Opaque](#)

```
func (p *RGBA64) Opaque() bool
```

`Opaque` scans the entire image and returns whether or not it is fully opaque.

func (*[RGBA64](#)) [PixOffset](#)

```
func (p *RGBA64) PixOffset(x, y int) int
```

PixOffset returns the index of the first element of Pix that corresponds to the pixel at (x, y).

func (*RGBA64) [Set](#)

```
func (p *RGBA64) Set(x, y int, c color.Color)
```

func (*RGBA64) [SetRGBA64](#)

```
func (p *RGBA64) SetRGBA64(x, y int, c color.RGBA64)
```

func (*RGBA64) [SubImage](#)

```
func (p *RGBA64) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

type [Rectangle](#)

```
type Rectangle struct {  
    Min, Max Point  
}
```

A Rectangle contains the points with $\text{Min.X} \leq X < \text{Max.X}$, $\text{Min.Y} \leq Y < \text{Max.Y}$. It is well-formed if $\text{Min.X} \leq \text{Max.X}$ and likewise for Y. Points are always well-formed. A rectangle's methods always return well-formed outputs for well-formed inputs.

```
var ZR Rectangle
```

ZR is the zero Rectangle.

func [Rect](#)

```
func Rect(x0, y0, x1, y1 int) Rectangle
```

Rect is shorthand for `Rectangle{Pt(x0, y0), Pt(x1, y1)}`.

func (Rectangle) [Add](#)

```
func (r Rectangle) Add(p Point) Rectangle
```

Add returns the rectangle r translated by p.

func (Rectangle) [Canon](#)

```
func (r Rectangle) Canon() Rectangle
```

Canon returns the canonical version of r. The returned rectangle has minimum and maximum coordinates swapped if necessary so that it is well-formed.

func (Rectangle) [Dx](#)

```
func (r Rectangle) Dx() int
```

Dx returns r's width.

func (Rectangle) [Dy](#)

```
func (r Rectangle) Dy() int
```

Dy returns r's height.

func (Rectangle) [Empty](#)

```
func (r Rectangle) Empty() bool
```

Empty returns whether the rectangle contains no points.

func (Rectangle) [Eq](#)

```
func (r Rectangle) Eq(s Rectangle) bool
```

Eq returns whether r and s are equal.

func (Rectangle) [In](#)

```
func (r Rectangle) In(s Rectangle) bool
```

In returns whether every point in r is in s.

func (Rectangle) [Inset](#)

```
func (r Rectangle) Inset(n int) Rectangle
```

Inset returns the rectangle r inset by n, which may be negative. If either of r's dimensions is less than $2*n$ then an empty rectangle near the center of r will be returned.

func (Rectangle) [Intersect](#)

```
func (r Rectangle) Intersect(s Rectangle) Rectangle
```

Intersect returns the largest rectangle contained by both r and s. If the two rectangles do not overlap then the zero rectangle will be returned.

func (Rectangle) [Overlaps](#)

```
func (r Rectangle) Overlaps(s Rectangle) bool
```

Overlaps returns whether r and s have a non-empty intersection.

func (Rectangle) [Size](#)

```
func (r Rectangle) Size() Point
```

Size returns r's width and height.

func (Rectangle) [String](#)

```
func (r Rectangle) String() string
```

String returns a string representation of r like "(3,4)-(6,5)".

func (Rectangle) [Sub](#)

```
func (r Rectangle) Sub(p Point) Rectangle
```

Sub returns the rectangle r translated by -p.

func (Rectangle) [Union](#)

```
func (r Rectangle) Union(s Rectangle) Rectangle
```

Union returns the smallest rectangle that contains both r and s.

type [Uniform](#)

```
type Uniform struct {  
    C color.Color  
}
```

Uniform is an infinite-sized Image of uniform color. It implements the `color.Color`, `color.ColorModel`, and `Image` interfaces.

func [NewUniform](#)

```
func NewUniform(c color.Color) *Uniform
```

func (*Uniform) [At](#)

```
func (c *Uniform) At(x, y int) color.Color
```

func (*Uniform) [Bounds](#)

```
func (c *Uniform) Bounds() Rectangle
```

func (*Uniform) [ColorModel](#)

```
func (c *Uniform) ColorModel() color.Model
```

func (*Uniform) [Convert](#)

```
func (c *Uniform) Convert(color.Color) color.Color
```

func (*Uniform) [Opaque](#)

```
func (c *Uniform) Opaque() bool
```

`Opaque` scans the entire image and returns whether or not it is fully opaque.

func (*Uniform) [RGBA](#)

```
func (c *Uniform) RGBA() (r, g, b, a uint32)
```

type [YCbCr](#)

```
type YCbCr struct {
    Y, Cb, Cr    []uint8
    YStride      int
    CStride      int
    SubsampleRatio YCbCrSubsampleRatio
    Rect         Rectangle
}
```

YCbCr is an in-memory image of Y'CbCr colors. There is one Y sample per pixel, but each Cb and Cr sample can span one or more pixels. YStride is the Y slice index delta between vertically adjacent pixels. CStride is the Cb and Cr slice index delta between vertically adjacent pixels that map to separate chroma samples. It is not an absolute requirement, but YStride and len(Y) are typically multiples of 8, and:

```
For 4:4:4, CStride == YStride/1 && len(Cb) == len(Cr) == len(Y)/1.
For 4:2:2, CStride == YStride/2 && len(Cb) == len(Cr) == len(Y)/2.
For 4:2:0, CStride == YStride/2 && len(Cb) == len(Cr) == len(Y)/4.
```

func [NewYCbCr](#)

```
func NewYCbCr(r Rectangle, subsampleRatio YCbCrSubsampleRatio) *YCbCr
```

NewYCbCr returns a new YCbCr with the given bounds and subsample ratio.

func (*YCbCr) [At](#)

```
func (p *YCbCr) At(x, y int) color.Color
```

func (*YCbCr) [Bounds](#)

```
func (p *YCbCr) Bounds() Rectangle
```

func (*YCbCr) [COffset](#)

```
func (p *YCbCr) COffset(x, y int) int
```

COffset returns the index of the first element of Cb or Cr that corresponds to the

pixel at (x, y).

func (*YCbCr) [ColorModel](#)

```
func (p *YCbCr) ColorModel() color.Model
```

func (*YCbCr) [Opaque](#)

```
func (p *YCbCr) Opaque() bool
```

func (*YCbCr) [SubImage](#)

```
func (p *YCbCr) SubImage(r Rectangle) Image
```

SubImage returns an image representing the portion of the image p visible through r. The returned value shares pixels with the original image.

func (*YCbCr) [YOffset](#)

```
func (p *YCbCr) YOffset(x, y int) int
```

YOffset returns the index of the first element of Y that corresponds to the pixel at (x, y).

type [YCbCrSubsampleRatio](#)

```
type YCbCrSubsampleRatio int
```

YCbCrSubsampleRatio is the chroma subsample ratio used in a YCbCr image.

```
const (  
    YCbCrSubsampleRatio444 YCbCrSubsampleRatio = iota  
    YCbCrSubsampleRatio422  
    YCbCrSubsampleRatio420  
)
```

func (YCbCrSubsampleRatio) [String](#)

```
func (s YCbCrSubsampleRatio) String() string
```

Subdirectories

Name	Synopsis
------	----------

color	Package color implements a basic color library.
-----------------------	---

draw	Package draw provides image composition functions.
----------------------	--

gif	Package gif implements a GIF image decoder.
---------------------	---

jpeg	Package jpeg implements a JPEG image decoder and encoder.
----------------------	---

png	Package png implements a PNG image decoder and encoder.
---------------------	---

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package color

```
import "image/color"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package color implements a basic color library.

Index

Variables

[func RGBToYCbCr\(r, g, b uint8\) \(uint8, uint8, uint8\)](#)

[func YCbCrToRGB\(y, cb, cr uint8\) \(uint8, uint8, uint8\)](#)

[type Alpha](#)

[func \(c Alpha\) RGBA\(\) \(r, g, b, a uint32\)](#)

[type Alpha16](#)

[func \(c Alpha16\) RGBA\(\) \(r, g, b, a uint32\)](#)

[type Color](#)

[type Gray](#)

[func \(c Gray\) RGBA\(\) \(r, g, b, a uint32\)](#)

[type Gray16](#)

[func \(c Gray16\) RGBA\(\) \(r, g, b, a uint32\)](#)

[type Model](#)

[func ModelFunc\(f func\(Color\) Color\) Model](#)

[type NRGBA](#)

[func \(c NRGBA\) RGBA\(\) \(r, g, b, a uint32\)](#)

[type NRGBA64](#)

[func \(c NRGBA64\) RGBA\(\) \(r, g, b, a uint32\)](#)

[type Palette](#)

[func \(p Palette\) Convert\(c Color\) Color](#)

[func \(p Palette\) Index\(c Color\) int](#)

[type RGBA](#)

[func \(c RGBA\) RGBA\(\) \(r, g, b, a uint32\)](#)

[type RGBA64](#)

[func \(c RGBA64\) RGBA\(\) \(r, g, b, a uint32\)](#)

[type YCbCr](#)

[func \(c YCbCr\) RGBA\(\) \(uint32, uint32, uint32, uint32\)](#)

Package files

[color.go](#) [ycbcr.go](#)

Variables

```
var (  
    Black      = Gray16{0}  
    White      = Gray16{0xffff}  
    Transparent = Alpha16{0}  
    Opaque     = Alpha16{0xffff}  
)
```

Standard colors.

func [RGBToYCbCr](#)

```
func RGBToYCbCr(r, g, b uint8) (uint8, uint8, uint8)
```

RGBToYCbCr converts an RGB triple to a Y'CbCr triple.

func YCbCrToRGB

```
func YCbCrToRGB(y, cb, cr uint8) (uint8, uint8, uint8)
```

YCbCrToRGB converts a Y'CbCr triple to an RGB triple.

type [Alpha](#)

```
type Alpha struct {  
    A uint8  
}
```

Alpha represents an 8-bit alpha color.

func (Alpha) [RGBA](#)

```
func (c Alpha) RGBA() (r, g, b, a uint32)
```

type [Alpha16](#)

```
type Alpha16 struct {  
    A uint16  
}
```

Alpha16 represents a 16-bit alpha color.

func (Alpha16) [RGBA](#)

```
func (c Alpha16) RGBA() (r, g, b, a uint32)
```

type [Color](#)

```
type Color interface {
    // RGBA returns the alpha-premultiplied red, green, blue and alp
    // for the color. Each value ranges within [0, 0xFFFF], but is r
    // by a uint32 so that multiplying by a blend factor up to 0xFFF
    // overflow.
    RGBA() (r, g, b, a uint32)
}
```

Color can convert itself to alpha-premultiplied 16-bits per channel RGBA. The conversion may be lossy.

type [Gray](#)

```
type Gray struct {  
    Y uint8  
}
```

Gray represents an 8-bit grayscale color.

func (Gray) [RGBA](#)

```
func (c Gray) RGBA() (r, g, b, a uint32)
```

type [Gray16](#)

```
type Gray16 struct {  
    Y uint16  
}
```

Gray16 represents a 16-bit grayscale color.

func (Gray16) [RGBA](#)

```
func (c Gray16) RGBA() (r, g, b, a uint32)
```

type [Model](#)

```
type Model interface {  
    Convert(c Color) Color  
}
```

Model can convert any Color to one from its own color model. The conversion may be lossy.

```
var (  
    RGBAModel    Model = ModelFunc(rgbaModel)  
    RGBA64Model  Model = ModelFunc(rgba64Model)  
    NRGBAModel   Model = ModelFunc(nrgbaModel)  
    NRGBA64Model Model = ModelFunc(nrgba64Model)  
    AlphaModel    Model = ModelFunc(alphaModel)  
    Alpha16Model  Model = ModelFunc(alpha16Model)  
    GrayModel     Model = ModelFunc(grayModel)  
    Gray16Model   Model = ModelFunc(gray16Model)  
)
```

Models for the standard color types.

```
var YCbCrModel Model = ModelFunc(yCbCrModel)
```

YCbCrModel is the Model for Y'CbCr colors.

func [ModelFunc](#)

```
func ModelFunc(f func(Color) Color) Model
```

ModelFunc returns a Model that invokes f to implement the conversion.

type [NRGBA](#)

```
type NRGBA struct {  
    R, G, B, A uint8  
}
```

NRGBA represents a non-alpha-premultiplied 32-bit color.

func (NRGBA) [RGBA](#)

```
func (c NRGBA) RGBA() (r, g, b, a uint32)
```

type [NRGBA64](#)

```
type NRGBA64 struct {  
    R, G, B, A uint16  
}
```

NRGBA64 represents a non-alpha-premultiplied 64-bit color, having 16 bits for each of red, green, blue and alpha.

func (NRGBA64) [RGBA](#)

```
func (c NRGBA64) RGBA() (r, g, b, a uint32)
```

type [Palette](#)

```
type Palette []Color
```

Palette is a palette of colors.

func (Palette) [Convert](#)

```
func (p Palette) Convert(c Color) Color
```

Convert returns the palette color closest to c in Euclidean R,G,B space.

func (Palette) [Index](#)

```
func (p Palette) Index(c Color) int
```

Index returns the index of the palette color closest to c in Euclidean R,G,B space.

type [RGBA](#)

```
type RGBA struct {  
    R, G, B, A uint8  
}
```

RGBA represents a traditional 32-bit alpha-premultiplied color, having 8 bits for each of red, green, blue and alpha.

func (RGBA) [RGBA](#)

```
func (c RGBA) RGBA() (r, g, b, a uint32)
```

type [RGBA64](#)

```
type RGBA64 struct {  
    R, G, B, A uint16  
}
```

RGBA64 represents a 64-bit alpha-premultiplied color, having 16 bits for each of red, green, blue and alpha.

func (RGBA64) [RGBA](#)

```
func (c RGBA64) RGBA() (r, g, b, a uint32)
```

type [YCbCr](#)

```
type YCbCr struct {  
    Y, Cb, Cr uint8  
}
```

YCbCr represents a fully opaque 24-bit Y'CbCr color, having 8 bits each for one luma and two chroma components.

JPEG, VP8, the MPEG family and other codecs use this color model. Such codecs often use the terms YUV and Y'CbCr interchangeably, but strictly speaking, the term YUV applies only to analog video signals, and Y' (luma) is Y (luminance) after applying gamma correction.

Conversion between RGB and Y'CbCr is lossy and there are multiple, slightly different formulae for converting between the two. This package follows the JFIF specification at <http://www.w3.org/Graphics/JPEG/jfif3.pdf>.

func (YCbCr) [RGBA](#)

```
func (c YCbCr) RGBA() (uint32, uint32, uint32, uint32)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package draw

```
import "image/draw"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package draw provides image composition functions.

See "The Go image/draw package" for an introduction to this package:
http://golang.org/doc/articles/image_draw.html

Index

[func Draw\(dst Image, r image.Rectangle, src image.Image, sp image.Point, op Op\)](#)

[func DrawMask\(dst Image, r image.Rectangle, src image.Image, sp image.Point, mask image.Image, mp image.Point, op Op\)](#)

[type Image](#)

[type Op](#)

Package files

[draw.go](#)

func [Draw](#)

```
func Draw(dst Image, r image.Rectangle, src image.Image, sp image.Po
```

Draw calls DrawMask with a nil mask.

func DrawMask

```
func DrawMask(dst Image, r image.Rectangle, src image.Image, sp imag
```

DrawMask aligns r.Min in dst with sp in src and mp in mask and then replaces the rectangle r in dst with the result of a Porter-Duff composition. A nil mask is treated as opaque.

type Image

```
type Image interface {  
    image.Image  
    Set(x, y int, c color.Color)  
}
```

A `draw.Image` is an `image.Image` with a `Set` method to change a single pixel.

type [Op](#)

```
type Op int
```

Op is a Porter-Duff compositing operator.

```
const (  
    // Over specifies ``(src in mask) over dst''.  
    Over Op = iota  
    // Src specifies ``src in mask''.  
    Src  
)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package gif

```
import "image/gif"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package gif implements a GIF image decoder.

The GIF specification is at <http://www.w3.org/Graphics/GIF/spec-gif89a.txt>.

Index

[func Decode\(r io.Reader\) \(image.Image, error\)](#)
[func DecodeConfig\(r io.Reader\) \(image.Config, error\)](#)
[type GIF](#)
[func DecodeAll\(r io.Reader\) \(*GIF, error\)](#)

Package files

[reader.go](#)

func Decode

```
func Decode(r io.Reader) (image.Image, error)
```

Decode reads a GIF image from r and returns the first embedded image as an image.Image.

func [DecodeConfig](#)

```
func DecodeConfig(r io.Reader) (image.Config, error)
```

DecodeConfig returns the global color model and dimensions of a GIF image without decoding the entire image.

type [GIF](#)

```
type GIF struct {
    Image      []*image.Paletted // The successive images.
    Delay      []int           // The successive delay times, one p
    LoopCount  int             // The loop count.
}
```

GIF represents the possibly multiple images stored in a GIF file.

func [DecodeAll](#)

```
func DecodeAll(r io.Reader) (*GIF, error)
```

DecodeAll reads a GIF image from r and returns the sequential frames and timing information.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package jpeg

```
import "image/jpeg"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package jpeg implements a JPEG image decoder and encoder.

JPEG is defined in ITU-T T.81: <http://www.w3.org/Graphics/JPEG/itu-t81.pdf>.

Index

Constants

[func Decode\(r io.Reader\) \(image.Image, error\)](#)

[func DecodeConfig\(r io.Reader\) \(image.Config, error\)](#)

[func Encode\(w io.Writer, m image.Image, o *Options\) error](#)

[type FormatError](#)

[func \(e FormatError\) Error\(\) string](#)

[type Options](#)

[type Reader](#)

[type UnsupportedError](#)

[func \(e UnsupportedError\) Error\(\) string](#)

Package files

[fdct.go](#) [huffman.go](#) [idct.go](#) [reader.go](#) [writer.go](#)

Constants

```
const DefaultQuality = 75
```

DefaultQuality is the default quality encoding parameter.

func [Decode](#)

```
func Decode(r io.Reader) (image.Image, error)
```

Decode reads a JPEG image from r and returns it as an image.Image.

func [DecodeConfig](#)

```
func DecodeConfig(r io.Reader) (image.Config, error)
```

DecodeConfig returns the color model and dimensions of a JPEG image without decoding the entire image.

func [Encode](#)

```
func Encode(w io.Writer, m image.Image, o *Options) error
```

Encode writes the Image m to w in JPEG 4:2:0 baseline format with the given options. Default parameters are used if a nil *Options is passed.

type [FormatError](#)

```
type FormatError string
```

A FormatError reports that the input is not a valid JPEG.

func (FormatError) [Error](#)

```
func (e FormatError) Error() string
```

type Options

```
type Options struct {  
    Quality int  
}
```

Options are the encoding parameters. Quality ranges from 1 to 100 inclusive, higher is better.

type Reader

```
type Reader interface {  
    io.Reader  
    ReadByte() (c byte, err error)  
}
```

If the passed in `io.Reader` does not also have `ReadByte`, then `Decode` will introduce its own buffering.

type [UnsupportedError](#)

```
type UnsupportedError string
```

An UnsupportedError reports that the input uses a valid but unimplemented JPEG feature.

func (UnsupportedError) [Error](#)

```
func (e UnsupportedError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package png

```
import "image/png"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package png implements a PNG image decoder and encoder.

The PNG specification is at <http://www.w3.org/TR/PNG/>.

Index

[func Decode\(r io.Reader\) \(image.Image, error\)](#)
[func DecodeConfig\(r io.Reader\) \(image.Config, error\)](#)
[func Encode\(w io.Writer, m image.Image\) error](#)
[type FormatError](#)
 [func \(e FormatError\) Error\(\) string](#)
[type UnsupportedError](#)
 [func \(e UnsupportedError\) Error\(\) string](#)

Package files

[reader.go](#) [writer.go](#)

func [Decode](#)

```
func Decode(r io.Reader) (image.Image, error)
```

Decode reads a PNG image from `r` and returns it as an `image.Image`. The type of `Image` returned depends on the PNG contents.

func [DecodeConfig](#)

```
func DecodeConfig(r io.Reader) (image.Config, error)
```

DecodeConfig returns the color model and dimensions of a PNG image without decoding the entire image.

func [Encode](#)

```
func Encode(w io.Writer, m image.Image) error
```

Encode writes the Image m to w in PNG format. Any Image may be encoded, but images that are not image.NRGBA might be encoded lossily.

type [FormatError](#)

```
type FormatError string
```

A FormatError reports that the input is not a valid PNG.

func (FormatError) [Error](#)

```
func (e FormatError) Error() string
```

type [UnsupportedError](#)

```
type UnsupportedError string
```

An `UnsupportedError` reports that the input uses a valid but unimplemented PNG feature.

func (UnsupportedError) [Error](#)

```
func (e UnsupportedError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg/index

Name	Synopsis
suffixarray	Package suffixarray implements substring search in logarithmic time using an in-memory suffix array.

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package suffixarray

```
import "index/suffixarray"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `suffixarray` implements substring search in logarithmic time using an in-memory suffix array.

Example use:

```
// create index for some data
index := suffixarray.New(data)

// lookup byte slice s
offsets1 := index.Lookup(s, -1) // the list of all indices where s occurs
offsets2 := index.Lookup(s, 3)  // the list of at most 3 indices where s occurs
```

Index

[type Index](#)

[func New\(data \[\]byte\) *Index](#)

[func \(x *Index\) Bytes\(\) \[\]byte](#)

[func \(x *Index\) FindAllIndex\(r *regexp.Regexp, n int\) \(result \[\]\[\]int\)](#)

[func \(x *Index\) Lookup\(s \[\]byte, n int\) \(result \[\]int\)](#)

[func \(x *Index\) Read\(r io.Reader\) error](#)

[func \(x *Index\) Write\(w io.Writer\) error](#)

Package files

[qsufsort.go](#) [suffixarray.go](#)

type [Index](#)

```
type Index struct {  
    // contains filtered or unexported fields  
}
```

Index implements a suffix array for fast substring search.

func [New](#)

```
func New(data []byte) *Index
```

New creates a new Index for data. Index creation time is $O(N \cdot \log(N))$ for $N = \text{len}(\text{data})$.

func (*Index) [Bytes](#)

```
func (x *Index) Bytes() []byte
```

Bytes returns the data over which the index was created. It must not be modified.

func (*Index) [FindAllIndex](#)

```
func (x *Index) FindAllIndex(r *regexp.Regexp, n int) (result [][]int)
```

FindAllIndex returns a sorted list of non-overlapping matches of the regular expression r , where a match is a pair of indices specifying the matched slice of $x.\text{Bytes}()$. If $n < 0$, all matches are returned in successive order. Otherwise, at most n matches are returned and they may not be successive. The result is nil if there are no matches, or if $n == 0$.

func (*Index) [Lookup](#)

```
func (x *Index) Lookup(s []byte, n int) (result []int)
```

Lookup returns an unsorted list of at most n indices where the byte string s occurs in the indexed data. If $n < 0$, all occurrences are returned. The result is nil if s is empty, s is not found, or $n == 0$. Lookup time is $O(\log(N) \cdot \text{len}(s) + \text{len}(\text{result}))$ where N is the size of the indexed data.

func (*Index) [Read](#)

```
func (x *Index) Read(r io.Reader) error
```

Read reads the index from r into x; x must not be nil.

func (*Index) [Write](#)

```
func (x *Index) Write(w io.Writer) error
```

Write writes the index x to w.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package io

```
import "io"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package `io` provides basic interfaces to I/O primitives. Its primary job is to wrap existing implementations of such primitives, such as those in package `os`, into shared public interfaces that abstract the functionality, plus some other related primitives.

Because these interfaces and primitives wrap lower-level operations with various implementations, unless otherwise informed clients should not assume they are safe for parallel execution.

Index

Variables

func Copy(dst Writer, src Reader) (written int64, err error)

func CopyN(dst Writer, src Reader, n int64) (written int64, err error)

func ReadAtLeast(r Reader, buf []byte, min int) (n int, err error)

func ReadFull(r Reader, buf []byte) (n int, err error)

func WriteString(w Writer, s string) (n int, err error)

type ByteReader

type ByteScanner

type Closer

type LimitedReader

func (l *LimitedReader) Read(p []byte) (n int, err error)

type PipeReader

func Pipe() (*PipeReader, *PipeWriter)

func (r *PipeReader) Close() error

func (r *PipeReader) CloseWithError(err error) error

func (r *PipeReader) Read(data []byte) (n int, err error)

type PipeWriter

func (w *PipeWriter) Close() error

func (w *PipeWriter) CloseWithError(err error) error

func (w *PipeWriter) Write(data []byte) (n int, err error)

type ReadCloser

type ReaderSeeker

type ReadWriteCloser

type ReadWriteSeeker

type ReadWriter

type Reader

func LimitReader(r Reader, n int64) Reader

func MultiReader(readers ...Reader) Reader

func TeeReader(r Reader, w Writer) Reader

type ReaderAt

type ReaderFrom

type RuneReader

type RuneScanner

type SectionReader

func NewSectionReader(r ReaderAt, off int64, n int64) *SectionReader

[func \(s *SectionReader\) Read\(p \[\]byte\) \(n int, err error\)](#)
[func \(s *SectionReader\) ReadAt\(p \[\]byte, off int64\) \(n int, err error\)](#)
[func \(s *SectionReader\) Seek\(offset int64, whence int\) \(ret int64, err error\)](#)
[func \(s *SectionReader\) Size\(\) int64](#)
[type Seeker](#)
[type WriteCloser](#)
[type WriteSeeker](#)
[type Writer](#)
[func MultiWriter\(writers ...Writer\) Writer](#)
[type WriterAt](#)
[type WriterTo](#)

Package files

[io.go](#) [multi.go](#) [pipe.go](#)

Variables

```
var EOF = errors.New("EOF")
```

EOF is the error returned by Read when no more input is available. Functions should return EOF only to signal a graceful end of input. If the EOF occurs unexpectedly in a structured data stream, the appropriate error is either ErrUnexpectedEOF or some other error giving more detail.

```
var ErrClosedPipe = errors.New("io: read/write on closed pipe")
```

ErrClosedPipe is the error used for read or write operations on a closed pipe.

```
var ErrShortBuffer = errors.New("short buffer")
```

ErrShortBuffer means that a read required a longer buffer than was provided.

```
var ErrShortWrite = errors.New("short write")
```

ErrShortWrite means that a write accepted fewer bytes than requested but failed to return an explicit error.

```
var ErrUnexpectedEOF = errors.New("unexpected EOF")
```

ErrUnexpectedEOF means that EOF was encountered in the middle of reading a fixed-size block or data structure.

func Copy

```
func Copy(dst Writer, src Reader) (written int64, err error)
```

Copy copies from src to dst until either EOF is reached on src or an error occurs. It returns the number of bytes copied and the first error encountered while copying, if any.

A successful Copy returns `err == nil`, not `err == EOF`. Because Copy is defined to read from src until EOF, it does not treat an EOF from Read as an error to be reported.

If dst implements the ReaderFrom interface, the copy is implemented by calling `dst.ReadFrom(src)`. Otherwise, if src implements the WriterTo interface, the copy is implemented by calling `src.WriteTo(dst)`.

func CopyN

`func CopyN(dst Writer, src Reader, n int64) (written int64, err error)`

CopyN copies n bytes (or until an error) from src to dst. It returns the number of bytes copied and the earliest error encountered while copying. Because Read can return the full amount requested as well as an error (including EOF), so can CopyN.

If dst implements the ReaderFrom interface, the copy is implemented using it.

func ReadAtLeast

```
func ReadAtLeast(r Reader, buf []byte, min int) (n int, err error)
```

ReadAtLeast reads from r into buf until it has read at least min bytes. It returns the number of bytes copied and an error if fewer bytes were read. The error is EOF only if no bytes were read. If an EOF happens after reading fewer than min bytes, ReadAtLeast returns ErrUnexpectedEOF. If min is greater than the length of buf, ReadAtLeast returns ErrShortBuffer.

func ReadFull

```
func ReadFull(r Reader, buf []byte) (n int, err error)
```

ReadFull reads exactly len(buf) bytes from r into buf. It returns the number of bytes copied and an error if fewer bytes were read. The error is EOF only if no bytes were read. If an EOF happens after reading some but not all the bytes, ReadFull returns ErrUnexpectedEOF.

func [WriteString](#)

```
func WriteString(w Writer, s string) (n int, err error)
```

WriteString writes the contents of the string `s` to `w`, which accepts an array of bytes. If `w` already implements a `WriteString` method, it is invoked directly.

type ByteReader

```
type ByteReader interface {  
    ReadByte() (c byte, err error)  
}
```

ByteReader is the interface that wraps the ReadByte method.

ReadByte reads and returns the next byte from the input. If no byte is available, err will be set.

type ByteScanner

```
type ByteScanner interface {  
    ByteReader  
    UnreadByte() error  
}
```

ByteScanner is the interface that adds the UnreadByte method to the basic ReadByte method.

UnreadByte causes the next call to ReadByte to return the same byte as the previous call to ReadByte. It may be an error to call UnreadByte twice without an intervening call to ReadByte.

type [Closer](#)

```
type Closer interface {  
    Close() error  
}
```

Closer is the interface that wraps the basic Close method.

type [LimitedReader](#)

```
type LimitedReader struct {  
    R Reader // underlying reader  
    N int64  // max bytes remaining  
}
```

A `LimitedReader` reads from `R` but limits the amount of data returned to just `N` bytes. Each call to `Read` updates `N` to reflect the new amount remaining.

func (***LimitedReader**) [Read](#)

```
func (l *LimitedReader) Read(p []byte) (n int, err error)
```

type [PipeReader](#)

```
type PipeReader struct {  
    // contains filtered or unexported fields  
}
```

A PipeReader is the read half of a pipe.

func [Pipe](#)

```
func Pipe() (*PipeReader, *PipeWriter)
```

Pipe creates a synchronous in-memory pipe. It can be used to connect code expecting an io.Reader with code expecting an io.Writer. Reads on one end are matched with writes on the other, copying data directly between the two; there is no internal buffering. It is safe to call Read and Write in parallel with each other or with Close. Close will complete once pending I/O is done. Parallel calls to Read, and parallel calls to Write, are also safe: the individual calls will be gated sequentially.

func (*PipeReader) [Close](#)

```
func (r *PipeReader) Close() error
```

Close closes the reader; subsequent writes to the write half of the pipe will return the error ErrClosedPipe.

func (*PipeReader) [CloseWithError](#)

```
func (r *PipeReader) CloseWithError(err error) error
```

CloseWithError closes the reader; subsequent writes to the write half of the pipe will return the error err.

func (*PipeReader) [Read](#)

```
func (r *PipeReader) Read(data []byte) (n int, err error)
```

Read implements the standard Read interface: it reads data from the pipe,

blocking until a writer arrives or the write end is closed. If the write end is closed with an error, that error is returned as err; otherwise err is EOF.

type [PipeWriter](#)

```
type PipeWriter struct {  
    // contains filtered or unexported fields  
}
```

A PipeWriter is the write half of a pipe.

func (*PipeWriter) [Close](#)

```
func (w *PipeWriter) Close() error
```

Close closes the writer; subsequent reads from the read half of the pipe will return no bytes and EOF.

func (*PipeWriter) [CloseWithError](#)

```
func (w *PipeWriter) CloseWithError(err error) error
```

CloseWithError closes the writer; subsequent reads from the read half of the pipe will return no bytes and the error err.

func (*PipeWriter) [Write](#)

```
func (w *PipeWriter) Write(data []byte) (n int, err error)
```

Write implements the standard Write interface: it writes data to the pipe, blocking until readers have consumed all the data or the read end is closed. If the read end is closed with an error, that err is returned as err; otherwise err is ErrClosedPipe.

type [ReadCloser](#)

```
type ReadCloser interface {  
    Reader  
    Closer  
}
```

ReadCloser is the interface that groups the basic Read and Close methods.

type [ReadSeeker](#)

```
type ReadSeeker interface {  
    Reader  
    Seeker  
}
```

ReadSeeker is the interface that groups the basic Read and Seek methods.

type [ReadWriteCloser](#)

```
type ReadWriteCloser interface {  
    Reader  
    Writer  
    Closer  
}
```

ReadWriteCloser is the interface that groups the basic Read, Write and Close methods.

type [ReadWriteSeeker](#)

```
type ReadWriteSeeker interface {  
    Reader  
    Writer  
    Seeker  
}
```

ReadWriteSeeker is the interface that groups the basic Read, Write and Seek methods.

type ReadWrite

```
type ReadWriter interface {  
    Reader  
    Writer  
}
```

ReadWrite is the interface that groups the basic Read and Write methods.

type [Reader](#)

```
type Reader interface {  
    Read(p []byte) (n int, err error)  
}
```

Reader is the interface that wraps the basic Read method.

Read reads up to len(p) bytes into p. It returns the number of bytes read (0 ≤ n ≤ len(p)) and any error encountered. Even if Read returns n < len(p), it may use all of p as scratch space during the call. If some data is available but not len(p) bytes, Read conventionally returns what is available instead of waiting for more.

When Read encounters an error or end-of-file condition after successfully reading n > 0 bytes, it returns the number of bytes read. It may return the (non-nil) error from the same call or return the error (and n == 0) from a subsequent call. An instance of this general case is that a Reader returning a non-zero number of bytes at the end of the input stream may return either err == EOF or err == nil. The next Read should return 0, EOF regardless.

Callers should always process the n > 0 bytes returned before considering the error err. Doing so correctly handles I/O errors that happen after reading some bytes and also both of the allowed EOF behaviors.

func [LimitReader](#)

```
func LimitReader(r Reader, n int64) Reader
```

LimitReader returns a Reader that reads from r but stops with EOF after n bytes. The underlying implementation is a *LimitedReader.

func [MultiReader](#)

```
func MultiReader(readers ...Reader) Reader
```

MultiReader returns a Reader that's the logical concatenation of the provided input readers. They're read sequentially. Once all inputs are drained, Read will return EOF.

func [TeeReader](#)

```
func TeeReader(r Reader, w Writer) Reader
```

TeeReader returns a Reader that writes to w what it reads from r. All reads from r performed through it are matched with corresponding writes to w. There is no internal buffering - the write must complete before the read completes. Any error encountered while writing is reported as a read error.

type [ReaderAt](#)

```
type ReaderAt interface {  
    ReadAt(p []byte, off int64) (n int, err error)  
}
```

ReaderAt is the interface that wraps the basic ReadAt method.

ReadAt reads len(p) bytes into p starting at offset off in the underlying input source. It returns the number of bytes read ($0 \leq n \leq \text{len}(p)$) and any error encountered.

When ReadAt returns $n < \text{len}(p)$, it returns a non-nil error explaining why more bytes were not returned. In this respect, ReadAt is stricter than Read.

Even if ReadAt returns $n < \text{len}(p)$, it may use all of p as scratch space during the call. If some data is available but not len(p) bytes, ReadAt blocks until either all the data is available or an error occurs. In this respect ReadAt is different from Read.

If the $n = \text{len}(p)$ bytes returned by ReadAt are at the end of the input source, ReadAt may return either `err == EOF` or `err == nil`.

If ReadAt is reading from an input source with a seek offset, ReadAt should not affect nor be affected by the underlying seek offset.

Clients of ReadAt can execute parallel ReadAt calls on the same input source.

type [ReaderFrom](#)

```
type ReaderFrom interface {  
    ReadFrom(r Reader) (n int64, err error)  
}
```

ReaderFrom is the interface that wraps the ReadFrom method.

type [RuneReader](#)

```
type RuneReader interface {  
    ReadRune() (r rune, size int, err error)  
}
```

RuneReader is the interface that wraps the ReadRune method.

ReadRune reads a single UTF-8 encoded Unicode character and returns the rune and its size in bytes. If no character is available, err will be set.

type [RuneScanner](#)

```
type RuneScanner interface {  
    RuneReader  
    UnreadRune() error  
}
```

RuneScanner is the interface that adds the UnreadRune method to the basic RuneReader method.

UnreadRune causes the next call to RuneReader to return the same rune as the previous call to RuneReader. It may be an error to call UnreadRune twice without an intervening call to RuneReader.

type [SectionReader](#)

```
type SectionReader struct {  
    // contains filtered or unexported fields  
}
```

SectionReader implements Read, Seek, and ReadAt on a section of an underlying ReaderAt.

func [NewSectionReader](#)

```
func NewSectionReader(r ReaderAt, off int64, n int64) *SectionReader
```

NewSectionReader returns a SectionReader that reads from r starting at offset off and stops with EOF after n bytes.

func (***SectionReader**) [Read](#)

```
func (s *SectionReader) Read(p []byte) (n int, err error)
```

func (***SectionReader**) [ReadAt](#)

```
func (s *SectionReader) ReadAt(p []byte, off int64) (n int, err error)
```

func (***SectionReader**) [Seek](#)

```
func (s *SectionReader) Seek(offset int64, whence int) (ret int64, err error)
```

func (***SectionReader**) [Size](#)

```
func (s *SectionReader) Size() int64
```

Size returns the size of the section in bytes.

type [Seeker](#)

```
type Seeker interface {  
    Seek(offset int64, whence int) (ret int64, err error)  
}
```

Seeker is the interface that wraps the basic Seek method.

Seek sets the offset for the next Read or Write to offset, interpreted according to whence: 0 means relative to the origin of the file, 1 means relative to the current offset, and 2 means relative to the end. Seek returns the new offset and an Error, if any.

type [WriteCloser](#)

```
type WriteCloser interface {  
    Writer  
    Closer  
}
```

WriteCloser is the interface that groups the basic Write and Close methods.

type [WriteSeeker](#)

```
type WriteSeeker interface {  
    Writer  
    Seeker  
}
```

WriteSeeker is the interface that groups the basic Write and Seek methods.

type [Writer](#)

```
type Writer interface {  
    Write(p []byte) (n int, err error)  
}
```

Writer is the interface that wraps the basic Write method.

Write writes len(p) bytes from p to the underlying data stream. It returns the number of bytes written from p (0 ≤ n ≤ len(p)) and any error encountered that caused the write to stop early. Write must return a non-nil error if it returns n < len(p).

func [MultiWriter](#)

```
func MultiWriter(writers ...Writer) Writer
```

MultiWriter creates a writer that duplicates its writes to all the provided writers, similar to the Unix tee(1) command.

type [WriterAt](#)

```
type WriterAt interface {  
    WriteAt(p []byte, off int64) (n int, err error)  
}
```

WriterAt is the interface that wraps the basic WriteAt method.

WriteAt writes len(p) bytes from p to the underlying data stream at offset off. It returns the number of bytes written from p (0 ≤ n ≤ len(p)) and any error encountered that caused the write to stop early. WriteAt must return a non-nil error if it returns n < len(p).

If WriteAt is writing to a destination with a seek offset, WriteAt should not affect nor be affected by the underlying seek offset.

Clients of WriteAt can execute parallel WriteAt calls on the same destination if the ranges do not overlap.

type [WriterTo](#)

```
type WriterTo interface {  
    WriteTo(w Writer) (n int64, err error)  
}
```

WriterTo is the interface that wraps the WriteTo method.

Subdirectories

Name **Synopsis**

[ioutil](#) Package ioutil implements some I/O utility functions.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package ioutil

```
import "io/ioutil"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package ioutil implements some I/O utility functions.

Index

Variables

[func NopCloser\(r io.Reader\) io.ReadCloser](#)

[func ReadAll\(r io.Reader\) \(\[\]byte, error\)](#)

[func ReadDir\(dirname string\) \(\[\]os.FileInfo, error\)](#)

[func ReadFile\(filename string\) \(\[\]byte, error\)](#)

[func TempDir\(dir, prefix string\) \(name string, err error\)](#)

[func TempFile\(dir, prefix string\) \(f *os.File, err error\)](#)

[func WriteFile\(filename string, data \[\]byte, perm os.FileMode\) error](#)

Package files

[ioutil.go](#) [tempfile.go](#)

Variables

```
var Discard io.Writer = devNull(0)
```

Discard is an `io.Writer` on which all `Write` calls succeed without doing anything.

func [NopCloser](#)

```
func NopCloser(r io.Reader) io.ReadCloser
```

NopCloser returns a ReadCloser with a no-op Close method wrapping the provided Reader r.

func [ReadAll](#)

```
func ReadAll(r io.Reader) ([]byte, error)
```

ReadAll reads from r until an error or EOF and returns the data it read. A successful call returns err == nil, not err == EOF. Because ReadAll is defined to read from src until EOF, it does not treat an EOF from Read as an error to be reported.

func [ReadDir](#)

```
func ReadDir(dirname string) ([]os.FileInfo, error)
```

ReadDir reads the directory named by dirname and returns a list of sorted directory entries.

func ReadFile

```
func ReadFile(filename string) ([]byte, error)
```

ReadFile reads the file named by filename and returns the contents. A successful call returns `err == nil`, not `err == EOF`. Because ReadFile reads the whole file, it does not treat an EOF from Read as an error to be reported.

func [TempDir](#)

```
func TempDir(dir, prefix string) (name string, err error)
```

TempDir creates a new temporary directory in the directory `dir` with a name beginning with `prefix` and returns the path of the new directory. If `dir` is the empty string, TempDir uses the default directory for temporary files (see `os.TempDir`). Multiple programs calling TempDir simultaneously will not choose the same directory. It is the caller's responsibility to remove the directory when no longer needed.

func [TempFile](#)

```
func TempFile(dir, prefix string) (f *os.File, err error)
```

TempFile creates a new temporary file in the directory dir with a name beginning with prefix, opens the file for reading and writing, and returns the resulting *os.File. If dir is the empty string, TempFile uses the default directory for temporary files (see os.TempDir). Multiple programs calling TempFile simultaneously will not choose the same file. The caller can use f.Name() to find the name of the file. It is the caller's responsibility to remove the file when no longer needed.

func [WriteFile](#)

```
func WriteFile(filename string, data []byte, perm os.FileMode) error
```

WriteFile writes data to a file named by filename. If the file does not exist, WriteFile creates it with permissions perm; otherwise WriteFile truncates it before writing.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package log

```
import "log"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package `log` implements a simple logging package. It defines a type, `Logger`, with methods for formatting output. It also has a predefined 'standard' `Logger` accessible through helper functions `Print[f|ln]`, `Fatal[f|ln]`, and `Panic[f|ln]`, which are easier to use than creating a `Logger` manually. That logger writes to standard error and prints the date and time of each logged message. The `Fatal` functions call `os.Exit(1)` after writing the log message. The `Panic` functions call `panic` after writing the log message.

Index

Constants

[func Fatal\(v ...interface{}\)](#)

[func Fata1f\(format string, v ...interface{}\)](#)

[func Fata1ln\(v ...interface{}\)](#)

[func Flags\(\) int](#)

[func Panic\(v ...interface{}\)](#)

[func Panicf\(format string, v ...interface{}\)](#)

[func Panicln\(v ...interface{}\)](#)

[func Prefix\(\) string](#)

[func Print\(v ...interface{}\)](#)

[func Printf\(format string, v ...interface{}\)](#)

[func Println\(v ...interface{}\)](#)

[func SetFlags\(flag int\)](#)

[func SetOutput\(w io.Writer\)](#)

[func SetPrefix\(prefix string\)](#)

[type Logger](#)

[func New\(out io.Writer, prefix string, flag int\) *Logger](#)

[func \(l *Logger\) Fatal\(v ...interface{}\)](#)

[func \(l *Logger\) Fata1f\(format string, v ...interface{}\)](#)

[func \(l *Logger\) Fata1ln\(v ...interface{}\)](#)

[func \(l *Logger\) Flags\(\) int](#)

[func \(l *Logger\) Output\(calldepth int, s string\) error](#)

[func \(l *Logger\) Panic\(v ...interface{}\)](#)

[func \(l *Logger\) Panicf\(format string, v ...interface{}\)](#)

[func \(l *Logger\) Panicln\(v ...interface{}\)](#)

[func \(l *Logger\) Prefix\(\) string](#)

[func \(l *Logger\) Print\(v ...interface{}\)](#)

[func \(l *Logger\) Printf\(format string, v ...interface{}\)](#)

[func \(l *Logger\) Println\(v ...interface{}\)](#)

[func \(l *Logger\) SetFlags\(flag int\)](#)

[func \(l *Logger\) SetPrefix\(prefix string\)](#)

Package files

[log.go](#)

Constants

```
const (  
    // Bits or'ed together to control what's printed. There is no co  
    // order they appear (the order listed here) or the format they  
    // described in the comments). A colon appears after these item  
    // 2009/0123 01:23:23.123123 /a/b/c/d.go:23: message  
    Ldate          = 1 << iota    // the date: 2009/01/23  
    Ltime          // the time: 01:23:23  
    Lmicroseconds  // microsecond resolution: 01:23:2  
    Llongfile      // full file name and line number:  
    Lshortfile     // final file name element and lin  
    LstdFlags      = Ldate | Ltime // initial values for the standard  
)
```

These flags define which text to prefix to each log entry generated by the
Logger.

func Fatal

```
func Fatal(v ...interface{})
```

Fatal is equivalent to Print() followed by a call to os.Exit(1).

func [Fatalf](#)

```
func Fatalf(format string, v ...interface{})
```

Fataf is equivalent to Printf() followed by a call to os.Exit(1).

func [Fataln](#)

```
func Fataln(v ...interface{})
```

Fataln is equivalent to `Println()` followed by a call to `os.Exit(1)`.

func [Flags](#)

```
func Flags() int
```

Flags returns the output flags for the standard logger.

func [Panic](#)

```
func Panic(v ...interface{})
```

Panic is equivalent to Print() followed by a call to panic().

func [Panicf](#)

```
func Panicf(format string, v ...interface{})
```

Panicf is equivalent to Printf() followed by a call to panic().

func [Panicln](#)

```
func Panicln(v ...interface{})
```

Panicln is equivalent to Println() followed by a call to panic().

func Prefix

```
func Prefix() string
```

Prefix returns the output prefix for the standard logger.

func [Print](#)

```
func Print(v ...interface{})
```

Print calls Output to print to the standard logger. Arguments are handled in the manner of `fmt.Print`.

func [Printf](#)

```
func Printf(format string, v ...interface{})
```

Printf calls Output to print to the standard logger. Arguments are handled in the manner of `fmt.Printf`.

func [Println](#)

```
func Println(v ...interface{})
```

Println calls Output to print to the standard logger. Arguments are handled in the manner of `fmt.Println`.

func [SetFlags](#)

```
func SetFlags(flag int)
```

SetFlags sets the output flags for the standard logger.

func [SetOutput](#)

```
func SetOutput(w io.Writer)
```

SetOutput sets the output destination for the standard logger.

func [SetPrefix](#)

```
func SetPrefix(prefix string)
```

SetPrefix sets the output prefix for the standard logger.

type [Logger](#)

```
type Logger struct {  
    // contains filtered or unexported fields  
}
```

A `Logger` represents an active logging object that generates lines of output to an `io.Writer`. Each logging operation makes a single call to the `Writer`'s `Write` method. A `Logger` can be used simultaneously from multiple goroutines; it guarantees to serialize access to the `Writer`.

func [New](#)

```
func New(out io.Writer, prefix string, flag int) *Logger
```

`New` creates a new `Logger`. The `out` variable sets the destination to which log data will be written. The `prefix` appears at the beginning of each generated log line. The `flag` argument defines the logging properties.

func (*Logger) [Fatal](#)

```
func (l *Logger) Fatal(v ...interface{})
```

`Fatal` is equivalent to `l.Print()` followed by a call to `os.Exit(1)`.

func (*Logger) [Fatalf](#)

```
func (l *Logger) Fatalf(format string, v ...interface{})
```

`Fatalf` is equivalent to `l.Printf()` followed by a call to `os.Exit(1)`.

func (*Logger) [Fatalln](#)

```
func (l *Logger) Fatalln(v ...interface{})
```

`Fatalln` is equivalent to `l.Println()` followed by a call to `os.Exit(1)`.

func (*Logger) [Flags](#)

```
func (l *Logger) Flags() int
```

Flags returns the output flags for the logger.

func (*Logger) [Output](#)

```
func (l *Logger) Output(calldepth int, s string) error
```

Output writes the output for a logging event. The string *s* contains the text to print after the prefix specified by the flags of the Logger. A newline is appended if the last character of *s* is not already a newline. Calldepth is used to recover the PC and is provided for generality, although at the moment on all pre-defined paths it will be 2.

func (*Logger) [Panic](#)

```
func (l *Logger) Panic(v ...interface{})
```

Panic is equivalent to `l.Print()` followed by a call to `panic()`.

func (*Logger) [Panicf](#)

```
func (l *Logger) Panicf(format string, v ...interface{})
```

Panicf is equivalent to `l.Printf()` followed by a call to `panic()`.

func (*Logger) [Panicln](#)

```
func (l *Logger) Panicln(v ...interface{})
```

Panicln is equivalent to `l.Println()` followed by a call to `panic()`.

func (*Logger) [Prefix](#)

```
func (l *Logger) Prefix() string
```

Prefix returns the output prefix for the logger.

func (*Logger) [Print](#)

```
func (l *Logger) Print(v ...interface{})
```

Print calls l.Output to print to the logger. Arguments are handled in the manner of fmt.Print.

func (*Logger) [Printf](#)

```
func (l *Logger) Printf(format string, v ...interface{})
```

Printf calls l.Output to print to the logger. Arguments are handled in the manner of fmt.Printf.

func (*Logger) [Println](#)

```
func (l *Logger) Println(v ...interface{})
```

Println calls l.Output to print to the logger. Arguments are handled in the manner of fmt.Println.

func (*Logger) [SetFlags](#)

```
func (l *Logger) SetFlags(flag int)
```

SetFlags sets the output flags for the logger.

func (*Logger) [SetPrefix](#)

```
func (l *Logger) SetPrefix(prefix string)
```

SetPrefix sets the output prefix for the logger.

Subdirectories

Name **Synopsis**

[syslog](#) Package syslog provides a simple interface to the system log service.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package syslog

```
import "log/syslog"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package syslog provides a simple interface to the system log service. It can send messages to the syslog daemon using UNIX domain sockets, UDP, or TCP connections.

Index

[func NewLogger\(p Priority, logFlag int\) \(*log.Logger, error\)](#)

[type Priority](#)

[type Writer](#)

[func Dial\(network, raddr string, priority Priority, prefix string\) \(w *Writer, err error\)](#)

[func New\(priority Priority, prefix string\) \(w *Writer, err error\)](#)

[func \(w *Writer\) Alert\(m string\) \(err error\)](#)

[func \(w *Writer\) Close\(\) error](#)

[func \(w *Writer\) Crit\(m string\) \(err error\)](#)

[func \(w *Writer\) Debug\(m string\) \(err error\)](#)

[func \(w *Writer\) Emerg\(m string\) \(err error\)](#)

[func \(w *Writer\) Err\(m string\) \(err error\)](#)

[func \(w *Writer\) Info\(m string\) \(err error\)](#)

[func \(w *Writer\) Notice\(m string\) \(err error\)](#)

[func \(w *Writer\) Warning\(m string\) \(err error\)](#)

[func \(w *Writer\) Write\(b \[\]byte\) \(int, error\)](#)

Package files

[syslog.go](#) [syslog_unix.go](#)

func [NewLogger](#)

```
func NewLogger(p Priority, logFlag int) (*log.Logger, error)
```

NewLogger creates a log.Logger whose output is written to the system log service with the specified priority. The logFlag argument is the flag set passed through to log.New to create the Logger.

type [Priority](#)

```
type Priority int
```

```
const (  
    // From /usr/include/sys/syslog.h.  
    // These are the same on Linux, BSD, and OS X.  
    LOG_EMERG Priority = iota  
    LOG_ALERT  
    LOG_CRIT  
    LOG_ERR  
    LOG_WARNING  
    LOG_NOTICE  
    LOG_INFO  
    LOG_DEBUG  
)
```

type [Writer](#)

```
type Writer struct {  
    // contains filtered or unexported fields  
}
```

A Writer is a connection to a syslog server.

func [Dial](#)

```
func Dial(network, raddr string, priority Priority, prefix string) (
```

Dial establishes a connection to a log daemon by connecting to address raddr on the network net. Each write to the returned writer sends a log message with the given priority and prefix.

func [New](#)

```
func New(priority Priority, prefix string) (w *Writer, err error)
```

New establishes a new connection to the system log daemon. Each write to the returned writer sends a log message with the given priority and prefix.

func (*Writer) [Alert](#)

```
func (w *Writer) Alert(m string) (err error)
```

Alert logs a message using the LOG_ALERT priority.

func (*Writer) [Close](#)

```
func (w *Writer) Close() error
```

func (*Writer) [Crit](#)

```
func (w *Writer) Crit(m string) (err error)
```

Crit logs a message using the LOG_CRIT priority.

func (*Writer) [Debug](#)

```
func (w *Writer) Debug(m string) (err error)
```

Debug logs a message using the LOG_DEBUG priority.

func (*Writer) [Emerg](#)

```
func (w *Writer) Emerg(m string) (err error)
```

Emerg logs a message using the LOG_EMERG priority.

func (*Writer) [Err](#)

```
func (w *Writer) Err(m string) (err error)
```

Err logs a message using the LOG_ERR priority.

func (*Writer) [Info](#)

```
func (w *Writer) Info(m string) (err error)
```

Info logs a message using the LOG_INFO priority.

func (*Writer) [Notice](#)

```
func (w *Writer) Notice(m string) (err error)
```

Notice logs a message using the LOG_NOTICE priority.

func (*Writer) [Warning](#)

```
func (w *Writer) Warning(m string) (err error)
```

Warning logs a message using the LOG_WARNING priority.

func (*Writer) [Write](#)

```
func (w *Writer) Write(b []byte) (int, error)
```

Write sends a log message to the syslog daemon.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package math

```
import "math"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package math provides basic constants and mathematical functions.

Index

Constants

[func Abs\(x float64\) float64](#)
[func Acos\(x float64\) float64](#)
[func Acosh\(x float64\) float64](#)
[func Asin\(x float64\) float64](#)
[func Asinh\(x float64\) float64](#)
[func Atan\(x float64\) float64](#)
[func Atan2\(y, x float64\) float64](#)
[func Atanh\(x float64\) float64](#)
[func Cbrt\(x float64\) float64](#)
[func Ceil\(x float64\) float64](#)
[func Copysign\(x, y float64\) float64](#)
[func Cos\(x float64\) float64](#)
[func Cosh\(x float64\) float64](#)
[func Dim\(x, y float64\) float64](#)
[func Erf\(x float64\) float64](#)
[func Erfc\(x float64\) float64](#)
[func Exp\(x float64\) float64](#)
[func Exp2\(x float64\) float64](#)
[func Expm1\(x float64\) float64](#)
[func Float32bits\(f float32\) uint32](#)
[func Float32frombits\(b uint32\) float32](#)
[func Float64bits\(f float64\) uint64](#)
[func Float64frombits\(b uint64\) float64](#)
[func Floor\(x float64\) float64](#)
[func Frexp\(f float64\) \(frac float64, exp int\)](#)
[func Gamma\(x float64\) float64](#)
[func Hypot\(p, q float64\) float64](#)
[func Ilogb\(x float64\) int](#)
[func Inf\(sign int\) float64](#)
[func IsInf\(f float64, sign int\) bool](#)
[func IsNaN\(f float64\) \(is bool\)](#)
[func J0\(x float64\) float64](#)
[func J1\(x float64\) float64](#)
[func Jn\(n int, x float64\) float64](#)

[func Ldexp\(frac float64, exp int\) float64](#)
[func Lgamma\(x float64\) \(lgamma float64, sign int\)](#)
[func Log\(x float64\) float64](#)
[func Log10\(x float64\) float64](#)
[func Log1p\(x float64\) float64](#)
[func Log2\(x float64\) float64](#)
[func Logb\(x float64\) float64](#)
[func Max\(x, y float64\) float64](#)
[func Min\(x, y float64\) float64](#)
[func Mod\(x, y float64\) float64](#)
[func Modf\(f float64\) \(int float64, frac float64\)](#)
[func NaN\(\) float64](#)
[func Nextafter\(x, y float64\) \(r float64\)](#)
[func Pow\(x, y float64\) float64](#)
[func Pow10\(e int\) float64](#)
[func Remainder\(x, y float64\) float64](#)
[func Signbit\(x float64\) bool](#)
[func Sin\(x float64\) float64](#)
[func Sincos\(x float64\) \(sin, cos float64\)](#)
[func Sinh\(x float64\) float64](#)
[func Sqrt\(x float64\) float64](#)
[func Tan\(x float64\) float64](#)
[func Tanh\(x float64\) float64](#)
[func Trunc\(x float64\) float64](#)
[func Y0\(x float64\) float64](#)
[func Y1\(x float64\) float64](#)
[func Yn\(n int, x float64\) float64](#)

Package files

[abs.go](#) [acosh.go](#) [asin.go](#) [asinh.go](#) [atan.go](#) [atan2.go](#) [atanh.go](#) [bits.go](#) [cbrt.go](#) [const.go](#) [copysign.go](#) [dim.go](#) [erf.go](#) [exp.go](#) [expm1.go](#) [floor.go](#) [frexp.go](#) [gamma.go](#) [hypot.go](#) [j0.go](#) [j1.go](#) [jn.go](#) [ldexp.go](#) [lgamma.go](#) [log.go](#) [log10.go](#) [log1p.go](#) [logb.go](#) [mod.go](#) [modf.go](#) [nextafter.go](#) [pow.go](#) [pow10.go](#) [remainder.go](#) [signbit.go](#) [sin.go](#) [sincos.go](#) [sinh.go](#) [sqrt.go](#) [tan.go](#) [tanh.go](#) [unsafe.go](#)

Constants

```
const (  
    E      = 2.71828182845904523536028747135266249775724709369995957496  
    Pi     = 3.14159265358979323846264338327950288419716939937510582097  
    Phi    = 1.61803398874989484820458683436563811772030917980576286213  
  
    Sqrt2  = 1.4142135623730950488016887242096980785696718753769480  
    SqrtE  = 1.6487212707001281468486507878141635716537761007101480  
    SqrtPi = 1.7724538509055160272981674833411451827975494561223871  
    SqrtPhi = 1.2720196495140689642524224617374914917156080418400962  
  
    Ln2    = 0.69314718055994530941723212145817656807550013436025525  
    Log2E  = 1 / Ln2  
    Ln10   = 2.30258509299404568401799145468436420760110148862877297  
    Log10E = 1 / Ln10  
)
```

Mathematical constants. Reference: <http://oeis.org/Axxxxxx>

```
const (  
    MaxFloat32      = 3.4028234663852885981170418348451692544  
    SmallestNonzeroFloat32 = 1.4012984643248170709237295832899161312  
  
    MaxFloat64      = 1.7976931348623157081452742373170435679  
    SmallestNonzeroFloat64 = 4.9406564584124654417656879286822137236  
)
```

Floating-point limit values. Max is the largest finite value representable by the type. SmallestNonzero is the smallest positive, non-zero value representable by the type.

```
const (  
    MaxInt8   = 1<<7 - 1  
    MinInt8   = -1 << 7  
    MaxInt16  = 1<<15 - 1  
    MinInt16  = -1 << 15  
    MaxInt32  = 1<<31 - 1  
    MinInt32  = -1 << 31  
    MaxInt64  = 1<<63 - 1  
    MinInt64  = -1 << 63  
    MaxUInt8  = 1<<8 - 1  
    MaxUInt16 = 1<<16 - 1  
    MaxUInt32 = 1<<32 - 1  
    MaxUInt64 = 1<<64 - 1
```

)

Integer limit values.

func [Abs](#)

```
func Abs(x float64) float64
```

Abs returns the absolute value of x.

Special cases are:

Abs(Inf) = +Inf

Abs(NaN) = NaN

func Acos

func Acos(x float64) float64

Acos returns the arccosine of x.

Special case is:

$\text{Acos}(x) = \text{NaN}$ if $x < -1$ or $x > 1$

func [Acosh](#)

func Acosh(x float64) float64

Acosh(x) calculates the inverse hyperbolic cosine of x.

Special cases are:

Acosh(+Inf) = +Inf

Acosh(x) = NaN if $x < 1$

Acosh(NaN) = NaN

func [Asin](#)

```
func Asin(x float64) float64
```

Asin returns the arcsine of x.

Special cases are:

$\text{Asin}(0) = 0$

$\text{Asin}(x) = \text{NaN}$ if $x < -1$ or $x > 1$

func [Asinh](#)

func Asinh(x float64) float64

Asinh(x) calculates the inverse hyperbolic sine of x.

Special cases are:

Asinh(0) = 0

Asinh(Inf) = Inf

Asinh(NaN) = NaN

func [Atan](#)

```
func Atan(x float64) float64
```

Atan returns the arctangent of x.

Special cases are:

$\text{Atan}(0) = 0$

$\text{Atan}(\text{Inf}) = \text{Pi}/2$

func [Atan2](#)

func Atan2(y, x float64) float64

Atan2 returns the arc tangent of y/x , using the signs of the two to determine the quadrant of the return value.

Special cases are (in order):

```
Atan2(y, NaN) = NaN
Atan2(NaN, x) = NaN
Atan2(+0, x>=0) = +0
Atan2(-0, x>=0) = -0
Atan2(+0, x<=-0) = +Pi
Atan2(-0, x<=-0) = -Pi
Atan2(y>0, 0) = +Pi/2
Atan2(y<0, 0) = -Pi/2
Atan2(+Inf, +Inf) = +Pi/4
Atan2(-Inf, +Inf) = -Pi/4
Atan2(+Inf, -Inf) = 3Pi/4
Atan2(-Inf, -Inf) = -3Pi/4
Atan2(y, +Inf) = 0
Atan2(y>0, -Inf) = +Pi
Atan2(y<0, -Inf) = -Pi
Atan2(+Inf, x) = +Pi/2
Atan2(-Inf, x) = -Pi/2
```

func [Atanh](#)

func Atanh(x float64) float64

Atanh(x) calculates the inverse hyperbolic tangent of x.

Special cases are:

Atanh(1) = +Inf

Atanh(0) = 0

Atanh(-1) = -Inf

Atanh(x) = NaN if $x < -1$ or $x > 1$

Atanh(NaN) = NaN

func [Cbrt](#)

```
func Cbrt(x float64) float64
```

Cbrt returns the cube root of its argument.

Special cases are:

$\text{Cbrt}(0) = 0$

$\text{Cbrt}(\text{Inf}) = \text{Inf}$

$\text{Cbrt}(\text{NaN}) = \text{NaN}$

func Ceil

```
func Ceil(x float64) float64
```

Ceil returns the least integer value greater than or equal to x.

Special cases are:

$\text{Ceil}(0) = 0$

$\text{Ceil}(\text{Inf}) = \text{Inf}$

$\text{Ceil}(\text{NaN}) = \text{NaN}$

func Copysign

```
func Copysign(x, y float64) float64
```

Copysign(x, y) returns a value with the magnitude of x and the sign of y.

func Cos

```
func Cos(x float64) float64
```

Cos returns the cosine of x.

Special cases are:

$\text{Cos}(\text{Inf}) = \text{NaN}$

$\text{Cos}(\text{NaN}) = \text{NaN}$

func Cosh

```
func Cosh(x float64) float64
```

Cosh returns the hyperbolic cosine of x.

Special cases are:

$\text{Cosh}(0) = 1$

$\text{Cosh}(\text{Inf}) = +\text{Inf}$

$\text{Cosh}(\text{NaN}) = \text{NaN}$

func Dim

```
func Dim(x, y float64) float64
```

Dim returns the maximum of x-y or 0.

Special cases are:

Dim(+Inf, +Inf) = NaN

Dim(-Inf, -Inf) = NaN

Dim(x, NaN) = Dim(NaN, x) = NaN

func [Erf](#)

```
func Erf(x float64) float64
```

Erf(x) returns the error function of x.

Special cases are:

Erf(+Inf) = 1

Erf(-Inf) = -1

Erf(NaN) = NaN

func [Erfc](#)

```
func Erfc(x float64) float64
```

Erfc(x) returns the complementary error function of x.

Special cases are:

$\text{Erfc}(+\text{Inf}) = 0$

$\text{Erfc}(-\text{Inf}) = 2$

$\text{Erfc}(\text{NaN}) = \text{NaN}$

func Exp

```
func Exp(x float64) float64
```

Exp returns $e^{**}x$, the base-e exponential of x.

Special cases are:

Exp(+Inf) = +Inf

Exp(NaN) = NaN

Very large values overflow to 0 or +Inf. Very small values underflow to 1.

func [Exp2](#)

```
func Exp2(x float64) float64
```

Exp2 returns $2^{**}x$, the base-2 exponential of x.

Special cases are the same as Exp.

func [Exp1](#)

```
func Exp1(x float64) float64
```

Exp1 returns $e^{**x} - 1$, the base-e exponential of x minus 1. It is more accurate than $\text{Exp}(x) - 1$ when x is near zero.

Special cases are:

```
Exp1(+Inf) = +Inf
```

```
Exp1(-Inf) = -1
```

```
Exp1(NaN) = NaN
```

Very large values overflow to -1 or +Inf.

func Float32bits

```
func Float32bits(f float32) uint32
```

Float32bits returns the IEEE 754 binary representation of f.

func [Float32frombits](#)

```
func Float32frombits(b uint32) float32
```

Float32frombits returns the floating point number corresponding to the IEEE 754 binary representation b.

func [Float64bits](#)

```
func Float64bits(f float64) uint64
```

Float64bits returns the IEEE 754 binary representation of f.

func [Float64frombits](#)

```
func Float64frombits(b uint64) float64
```

Float64frombits returns the floating point number corresponding the IEEE 754 binary representation b.

func Floor

```
func Floor(x float64) float64
```

Floor returns the greatest integer value less than or equal to x.

Special cases are:

Floor(0) = 0

Floor(Inf) = Inf

Floor(NaN) = NaN

func Frexp

func Frexp(f float64) (frac float64, exp int)

Frexp breaks f into a normalized fraction and an integral power of two. It returns $frac$ and exp satisfying $f == frac \cdot 2^{**}exp$, with the absolute value of $frac$ in the interval $[?, 1)$.

Special cases are:

Frexp(0) = 0, 0

Frexp(Inf) = Inf, 0

Frexp(NaN) = NaN, 0

func Gamma

```
func Gamma(x float64) float64
```

Gamma(x) returns the Gamma function of x.

Special cases are:

Gamma(Inf) = Inf

Gamma(NaN) = NaN

Large values overflow to +Inf. Zero and negative integer arguments return Inf.

func Hypot

```
func Hypot(p, q float64) float64
```

Hypot computes $\text{Sqrt}(p^2 + q^2)$, taking care to avoid unnecessary overflow and underflow.

Special cases are:

Hypot(p, q) = +Inf if p or q is infinite

Hypot(p, q) = NaN if p or q is NaN

func [Ilogb](#)

```
func Ilogb(x float64) int
```

Ilogb(x) returns the binary exponent of x as an integer.

Special cases are:

```
Ilogb(Inf) = MaxInt32
```

```
Ilogb(0) = MinInt32
```

```
Ilogb(NaN) = MaxInt32
```

func Inf

```
func Inf(sign int) float64
```

Inf returns positive infinity if $\text{sign} \geq 0$, negative infinity if $\text{sign} < 0$.

func [IsInf](#)

```
func IsInf(f float64, sign int) bool
```

IsInf returns whether f is an infinity, according to sign. If sign > 0, IsInf returns whether f is positive infinity. If sign < 0, IsInf returns whether f is negative infinity. If sign == 0, IsInf returns whether f is either infinity.

func [IsNaN](#)

```
func IsNaN(f float64) (is bool)
```

IsNaN returns whether f is an IEEE 754 “not-a-number” value.

func [J0](#)

func J0(x float64) float64

J0 returns the order-zero Bessel function of the first kind.

Special cases are:

$J_0(\text{Inf}) = 0$

$J_0(0) = 1$

$J_0(\text{NaN}) = \text{NaN}$

func [J1](#)

func J1(x float64) float64

J1 returns the order-one Bessel function of the first kind.

Special cases are:

$J1(\text{Inf}) = 0$

$J1(\text{NaN}) = \text{NaN}$

func [Jn](#)

```
func Jn(n int, x float64) float64
```

Jn returns the order-n Bessel function of the first kind.

Special cases are:

$$J_n(n, \text{Inf}) = 0$$
$$J_n(n, \text{NaN}) = \text{NaN}$$

func Ldexp

func Ldexp(frac float64, exp int) float64

Ldexp is the inverse of Frexp. It returns $\text{frac} \cdot 2^{**}\text{exp}$.

Special cases are:

Ldexp(0, exp) = 0

Ldexp(Inf, exp) = Inf

Ldexp(NaN, exp) = NaN

func Lgamma

```
func Lgamma(x float64) (lgamma float64, sign int)
```

Lgamma returns the natural logarithm and sign (-1 or +1) of Gamma(x).

Special cases are:

```
Lgamma(+Inf) = +Inf  
Lgamma(0) = +Inf  
Lgamma(-integer) = +Inf  
Lgamma(-Inf) = -Inf  
Lgamma(NaN) = NaN
```

func **Log**

```
func Log(x float64) float64
```

Log returns the natural logarithm of x.

Special cases are:

$\text{Log}(+\text{Inf}) = +\text{Inf}$

$\text{Log}(0) = -\text{Inf}$

$\text{Log}(x < 0) = \text{NaN}$

$\text{Log}(\text{NaN}) = \text{NaN}$

func Log10

```
func Log10(x float64) float64
```

Log10 returns the decimal logarithm of x. The special cases are the same as for Log.

func [Log1p](#)

```
func Log1p(x float64) float64
```

Log1p returns the natural logarithm of 1 plus its argument x. It is more accurate than $\text{Log}(1 + x)$ when x is near zero.

Special cases are:

```
Log1p(+Inf) = +Inf  
Log1p(0) = 0  
Log1p(-1) = -Inf  
Log1p(x < -1) = NaN  
Log1p(NaN) = NaN
```

func Log2

```
func Log2(x float64) float64
```

Log2 returns the binary logarithm of x. The special cases are the same as for Log.

func Logb

```
func Logb(x float64) float64
```

Logb(x) returns the binary exponent of x.

Special cases are:

Logb(Inf) = +Inf

Logb(0) = -Inf

Logb(NaN) = NaN

func Max

```
func Max(x, y float64) float64
```

Max returns the larger of x or y.

Special cases are:

$\text{Max}(x, +\text{Inf}) = \text{Max}(+\text{Inf}, x) = +\text{Inf}$

$\text{Max}(x, \text{NaN}) = \text{Max}(\text{NaN}, x) = \text{NaN}$

$\text{Max}(+0, 0) = \text{Max}(0, +0) = +0$

$\text{Max}(-0, -0) = -0$

func Min

```
func Min(x, y float64) float64
```

Min returns the smaller of x or y.

Special cases are:

$\text{Min}(x, -\text{Inf}) = \text{Min}(-\text{Inf}, x) = -\text{Inf}$

$\text{Min}(x, \text{NaN}) = \text{Min}(\text{NaN}, x) = \text{NaN}$

$\text{Min}(-0, 0) = \text{Min}(0, -0) = -0$

func Mod

```
func Mod(x, y float64) float64
```

Mod returns the floating-point remainder of x/y . The magnitude of the result is less than y and its sign agrees with that of x .

Special cases are:

$\text{Mod}(\text{Inf}, y) = \text{NaN}$

$\text{Mod}(\text{NaN}, y) = \text{NaN}$

$\text{Mod}(x, 0) = \text{NaN}$

$\text{Mod}(x, \text{Inf}) = x$

$\text{Mod}(x, \text{NaN}) = \text{NaN}$

func Modf

```
func Modf(f float64) (int float64, frac float64)
```

Modf returns integer and fractional floating-point numbers that sum to f. Both values have the same sign as f.

Special cases are:

$\text{Modf}(\text{Inf}) = \text{Inf}, \text{NaN}$

$\text{Modf}(\text{NaN}) = \text{NaN}, \text{NaN}$

func [NaN](#)

```
func NaN() float64
```

NaN returns an IEEE 754 “not-a-number” value.

func [Nextafter](#)

```
func Nextafter(x, y float64) (r float64)
```

Nextafter returns the next representable value after x towards y. If x == y, then x is returned.

Special cases are:

```
Nextafter(NaN, y) = NaN
```

```
Nextafter(x, NaN) = NaN
```

func Pow

```
func Pow(x, y float64) float64
```

Pow returns $x^{**}y$, the base-x exponential of y.

Special cases are (in order):

$\text{Pow}(x, 0) = 1$ for any x

$\text{Pow}(1, y) = 1$ for any y

$\text{Pow}(x, 1) = x$ for any x

$\text{Pow}(\text{NaN}, y) = \text{NaN}$

$\text{Pow}(x, \text{NaN}) = \text{NaN}$

$\text{Pow}(0, y) = \text{Inf}$ for y an odd integer < 0

$\text{Pow}(0, -\text{Inf}) = +\text{Inf}$

$\text{Pow}(0, +\text{Inf}) = +0$

$\text{Pow}(0, y) = +\text{Inf}$ for finite $y < 0$ and not an odd integer

$\text{Pow}(0, y) = 0$ for y an odd integer > 0

$\text{Pow}(0, y) = +0$ for finite $y > 0$ and not an odd integer

$\text{Pow}(-1, \text{Inf}) = 1$

$\text{Pow}(x, +\text{Inf}) = +\text{Inf}$ for $|x| > 1$

$\text{Pow}(x, -\text{Inf}) = +0$ for $|x| > 1$

$\text{Pow}(x, +\text{Inf}) = +0$ for $|x| < 1$

$\text{Pow}(x, -\text{Inf}) = +\text{Inf}$ for $|x| < 1$

$\text{Pow}(+\text{Inf}, y) = +\text{Inf}$ for $y > 0$

$\text{Pow}(+\text{Inf}, y) = +0$ for $y < 0$

$\text{Pow}(-\text{Inf}, y) = \text{Pow}(-0, -y)$

$\text{Pow}(x, y) = \text{NaN}$ for finite $x < 0$ and finite non-integer y

func Pow10

```
func Pow10(e int) float64
```

Pow10 returns $10^{**}e$, the base-10 exponential of e.

Special cases are:

$\text{Pow10}(e) = +\text{Inf}$ for $e > 309$

$\text{Pow10}(e) = 0$ for $e < -324$

func Remainder

```
func Remainder(x, y float64) float64
```

Remainder returns the IEEE 754 floating-point remainder of x/y.

Special cases are:

Remainder(Inf, y) = NaN

Remainder(NaN, y) = NaN

Remainder(x, 0) = NaN

Remainder(x, Inf) = x

Remainder(x, NaN) = NaN

func [Signbit](#)

```
func Signbit(x float64) bool
```

Signbit returns true if x is negative or negative zero.

func [Sin](#)

```
func Sin(x float64) float64
```

Sin returns the sine of x.

Special cases are:

$\text{Sin}(0) = 0$

$\text{Sin}(\text{Inf}) = \text{NaN}$

$\text{Sin}(\text{NaN}) = \text{NaN}$

func Sincos

func Sincos(x float64) (sin, cos float64)

Sincos(x) returns Sin(x), Cos(x).

Special cases are:

Sincos(0) = 0, 1

Sincos(Inf) = NaN, NaN

Sincos(NaN) = NaN, NaN

func [Sinh](#)

```
func Sinh(x float64) float64
```

Sinh returns the hyperbolic sine of x.

Special cases are:

$\text{Sinh}(0) = 0$

$\text{Sinh}(\text{Inf}) = \text{Inf}$

$\text{Sinh}(\text{NaN}) = \text{NaN}$

func [Sqrt](#)

```
func Sqrt(x float64) float64
```

Sqrt returns the square root of x.

Special cases are:

$\text{Sqrt}(+\text{Inf}) = +\text{Inf}$

$\text{Sqrt}(0) = 0$

$\text{Sqrt}(x < 0) = \text{NaN}$

$\text{Sqrt}(\text{NaN}) = \text{NaN}$

func Tan

```
func Tan(x float64) float64
```

Tan returns the tangent of x.

Special cases are:

$\text{Tan}(0) = 0$

$\text{Tan}(\text{Inf}) = \text{NaN}$

$\text{Tan}(\text{NaN}) = \text{NaN}$

func [Tanh](#)

```
func Tanh(x float64) float64
```

Tanh computes the hyperbolic tangent of x.

Special cases are:

$$\text{Tanh}(0) = 0$$

$$\text{Tanh}(\text{Inf}) = 1$$

$$\text{Tanh}(\text{NaN}) = \text{NaN}$$

func Trunc

```
func Trunc(x float64) float64
```

Trunc returns the integer value of x.

Special cases are:

Trunc(0) = 0

Trunc(Inf) = Inf

Trunc(NaN) = NaN

func [Y0](#)

```
func Y0(x float64) float64
```

Y0 returns the order-zero Bessel function of the second kind.

Special cases are:

$$Y_0(+\text{Inf}) = 0$$

$$Y_0(0) = -\text{Inf}$$

$$Y_0(x < 0) = \text{NaN}$$

$$Y_0(\text{NaN}) = \text{NaN}$$

func [Y1](#)

```
func Y1(x float64) float64
```

Y1 returns the order-one Bessel function of the second kind.

Special cases are:

$$Y1(+Inf) = 0$$

$$Y1(0) = -Inf$$

$$Y1(x < 0) = NaN$$

$$Y1(NaN) = NaN$$

func [Yn](#)

```
func Yn(n int, x float64) float64
```

Yn returns the order-n Bessel function of the second kind.

Special cases are:

$Y_n(n, +\text{Inf}) = 0$

$Y_n(n > 0, 0) = -\text{Inf}$

$Y_n(n < 0, 0) = +\text{Inf}$ if n is odd, $-\text{Inf}$ if n is even

$Y_1(n, x < 0) = \text{NaN}$

$Y_1(n, \text{NaN}) = \text{NaN}$

Subdirectories

Name	Synopsis
------	----------

big	Package big implements multi-precision arithmetic (big numbers).
---------------------	--

cmplx	Package cmplx provides basic constants and mathematical functions for complex numbers.
-----------------------	--

rand	Package rand implements pseudo-random number generators.
----------------------	--

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package big

```
import "math/big"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package `big` implements multi-precision arithmetic (big numbers). The following numeric types are supported:

- `Int` signed integers
- `Rat` rational numbers

Methods are typically of the form:

```
func (z *Int) Op(x, y *Int) *Int          (similar for *Rat)
```

and implement operations $z = x \text{ Op } y$ with the result as receiver; if it is one of the operands it may be overwritten (and its memory reused). To enable chaining of operations, the result is also returned. Methods returning a result other than `*Int` or `*Rat` take one of the operands as the receiver.

Index

Constants

type Int

func NewInt(x int64) *Int
func (z *Int) Abs(x *Int) *Int
func (z *Int) Add(x, y *Int) *Int
func (z *Int) And(x, y *Int) *Int
func (z *Int) AndNot(x, y *Int) *Int
func (z *Int) Binomial(n, k int64) *Int
func (x *Int) Bit(i int) uint
func (x *Int) BitLen() int
func (x *Int) Bits() []Word
func (x *Int) Bytes() []byte
func (x *Int) Cmp(y *Int) (r int)
func (z *Int) Div(x, y *Int) *Int
func (z *Int) DivMod(x, y, m *Int) (*Int, *Int)
func (z *Int) Exp(x, y, m *Int) *Int
func (x *Int) Format(s fmt.State, ch rune)
func (z *Int) GCD(x, y, a, b *Int) *Int
func (z *Int) GobDecode(buf []byte) error
func (x *Int) GobEncode() ([]byte, error)
func (x *Int) Int64() int64
func (z *Int) Lsh(x *Int, n uint) *Int
func (z *Int) Mod(x, y *Int) *Int
func (z *Int) ModInverse(g, p *Int) *Int
func (z *Int) Mul(x, y *Int) *Int
func (z *Int) MulRange(a, b int64) *Int
func (z *Int) Neg(x *Int) *Int
func (z *Int) Not(x *Int) *Int
func (z *Int) Or(x, y *Int) *Int
func (x *Int) ProbablyPrime(n int) bool
func (z *Int) Quo(x, y *Int) *Int
func (z *Int) QuoRem(x, y, r *Int) (*Int, *Int)
func (z *Int) Rand(rnd *rand.Rand, n *Int) *Int
func (z *Int) Rem(x, y *Int) *Int
func (z *Int) Rsh(x *Int, n uint) *Int

func (z *Int) Scan(s fmt.ScanState, ch rune) error
func (z *Int) Set(x *Int) *Int
func (z *Int) SetBit(x *Int, i int, b uint) *Int
func (z *Int) SetBits(abs []Word) *Int
func (z *Int) SetBytes(buf []byte) *Int
func (z *Int) SetInt64(x int64) *Int
func (z *Int) SetString(s string, base int) (*Int, bool)
func (x *Int) Sign() int
func (x *Int) String() string
func (z *Int) Sub(x, y *Int) *Int
func (z *Int) Xor(x, y *Int) *Int

type Rat

func NewRat(a, b int64) *Rat
func (z *Rat) Abs(x *Rat) *Rat
func (z *Rat) Add(x, y *Rat) *Rat
func (x *Rat) Cmp(y *Rat) int
func (x *Rat) Denom() *Int
func (x *Rat) FloatString(prec int) string
func (z *Rat) GobDecode(buf []byte) error
func (x *Rat) GobEncode() ([]byte, error)
func (z *Rat) Inv(x *Rat) *Rat
func (x *Rat) IsInt() bool
func (z *Rat) Mul(x, y *Rat) *Rat
func (z *Rat) Neg(x *Rat) *Rat
func (x *Rat) Num() *Int
func (z *Rat) Quo(x, y *Rat) *Rat
func (x *Rat) RatString() string
func (z *Rat) Scan(s fmt.ScanState, ch rune) error
func (z *Rat) Set(x *Rat) *Rat
func (z *Rat) SetFrac(a, b *Int) *Rat
func (z *Rat) SetFrac64(a, b int64) *Rat
func (z *Rat) SetInt(x *Int) *Rat
func (z *Rat) SetInt64(x int64) *Rat
func (z *Rat) SetString(s string) (*Rat, bool)
func (x *Rat) Sign() int
func (x *Rat) String() string
func (z *Rat) Sub(x, y *Rat) *Rat

type Word

Examples

[Int.Scan](#)

[Int.SetString](#)

[Rat.Scan](#)

[Rat.SetString](#)

Package files

[arith.go](#) [arith_decl.go](#) [int.go](#) [nat.go](#) [rat.go](#)

Constants

```
const MaxBase = 'z' - 'a' + 10 + 1 // = hexValue('z') + 1
```

MaxBase is the largest number base accepted for string conversions.

type [Int](#)

```
type Int struct {  
    // contains filtered or unexported fields  
}
```

An Int represents a signed multi-precision integer. The zero value for an Int represents the value 0.

func [NewInt](#)

```
func NewInt(x int64) *Int
```

NewInt allocates and returns a new Int set to x.

func ([*Int](#)) [Abs](#)

```
func (z *Int) Abs(x *Int) *Int
```

Abs sets z to $|x|$ (the absolute value of x) and returns z.

func ([*Int](#)) [Add](#)

```
func (z *Int) Add(x, y *Int) *Int
```

Add sets z to the sum $x+y$ and returns z.

func ([*Int](#)) [And](#)

```
func (z *Int) And(x, y *Int) *Int
```

And sets $z = x \& y$ and returns z.

func ([*Int](#)) [AndNot](#)

```
func (z *Int) AndNot(x, y *Int) *Int
```

AndNot sets $z = x \&^{\wedge} y$ and returns z.

func (*Int) [Binomial](#)

```
func (z *Int) Binomial(n, k int64) *Int
```

Binomial sets z to the binomial coefficient of (n, k) and returns z.

func (*Int) [Bit](#)

```
func (x *Int) Bit(i int) uint
```

Bit returns the value of the i'th bit of x. That is, it returns $(x \gg i) \& 1$. The bit index i must be ≥ 0 .

func (*Int) [BitLen](#)

```
func (x *Int) BitLen() int
```

BitLen returns the length of the absolute value of z in bits. The bit length of 0 is 0.

func (*Int) [Bits](#)

```
func (x *Int) Bits() []Word
```

Bits provides raw (unchecked but fast) access to x by returning its absolute value as a little-endian Word slice. The result and x share the same underlying array. Bits is intended to support implementation of missing low-level Int functionality outside this package; it should be avoided otherwise.

func (*Int) [Bytes](#)

```
func (x *Int) Bytes() []byte
```

Bytes returns the absolute value of z as a big-endian byte slice.

func (*Int) [Cmp](#)

```
func (x *Int) Cmp(y *Int) (r int)
```

Cmp compares x and y and returns:

```
-1 if x < y
 0 if x == y
+1 if x > y
```

func (*Int) [Div](#)

```
func (z *Int) Div(x, y *Int) *Int
```

Div sets z to the quotient x/y for y != 0 and returns z. If y == 0, a division-by-zero run-time panic occurs. Div implements Euclidean division (unlike Go); see DivMod for more details.

func (*Int) [DivMod](#)

```
func (z *Int) DivMod(x, y, m *Int) (*Int, *Int)
```

DivMod sets z to the quotient x div y and m to the modulus x mod y and returns the pair (z, m) for y != 0. If y == 0, a division-by-zero run-time panic occurs.

DivMod implements Euclidean division and modulus (unlike Go):

$$q = x \operatorname{div} y \quad \text{such that}$$
$$m = x - y * q \quad \text{with } 0 \leq m < |q|$$

(See Raymond T. Boute, “The Euclidean definition of the functions div and mod”. ACM Transactions on Programming Languages and Systems (TOPLAS), 14(2):127-144, New York, NY, USA, 4/1992. ACM press.) See QuoRem for T-division and modulus (like Go).

func (*Int) [Exp](#)

```
func (z *Int) Exp(x, y, m *Int) *Int
```

Exp sets z = x**y mod m and returns z. If m is nil, z = x**y. See Knuth, volume 2, section 4.6.3.

func (*Int) [Format](#)

```
func (x *Int) Format(s fmt.State, ch rune)
```

Format is a support routine for fmt.Formatter. It accepts the formats 'b' (binary),

'o' (octal), 'd' (decimal), 'x' (lowercase hexadecimal), and 'X' (uppercase hexadecimal). Also supported are the full suite of package `fmt`'s format verbs for integral types, including '+', '-', and ' ' for sign control, '#' for leading zero in octal and for hexadecimal, a leading "0x" or "0X" for "%#x" and "%#X" respectively, specification of minimum digits precision, output field width, space or zero padding, and left or right justification.

func (*Int) [GCD](#)

```
func (z *Int) GCD(x, y, a, b *Int) *Int
```

`GCD` sets `z` to the greatest common divisor of `a` and `b`, which must be positive numbers, and returns `z`. If `x` and `y` are not nil, `GCD` sets `x` and `y` such that $z = a*x + b*y$. If either `a` or `b` is not positive, `GCD` sets $z = x = y = 0$.

func (*Int) [GobDecode](#)

```
func (z *Int) GobDecode(buf []byte) error
```

`GobDecode` implements the `gob.GobDecoder` interface.

func (*Int) [GobEncode](#)

```
func (x *Int) GobEncode() ([]byte, error)
```

`GobEncode` implements the `gob.GobEncoder` interface.

func (*Int) [Int64](#)

```
func (x *Int) Int64() int64
```

`Int64` returns the `int64` representation of `x`. If `x` cannot be represented in an `int64`, the result is undefined.

func (*Int) [Lsh](#)

```
func (z *Int) Lsh(x *Int, n uint) *Int
```

`Lsh` sets $z = x \ll n$ and returns `z`.

func (*Int) [Mod](#)

```
func (z *Int) Mod(x, y *Int) *Int
```

Mod sets z to the modulus $x\%y$ for $y \neq 0$ and returns z. If $y == 0$, a division-by-zero run-time panic occurs. Mod implements Euclidean modulus (unlike Go); see DivMod for more details.

func (*Int) [ModInverse](#)

```
func (z *Int) ModInverse(g, p *Int) *Int
```

ModInverse sets z to the multiplicative inverse of g in the group $\mathbb{Z}/p\mathbb{Z}$ (where p is a prime) and returns z.

func (*Int) [Mul](#)

```
func (z *Int) Mul(x, y *Int) *Int
```

Mul sets z to the product $x*y$ and returns z.

func (*Int) [MulRange](#)

```
func (z *Int) MulRange(a, b int64) *Int
```

MulRange sets z to the product of all integers in the range $[a, b]$ inclusively and returns z. If $a > b$ (empty range), the result is 1.

func (*Int) [Neg](#)

```
func (z *Int) Neg(x *Int) *Int
```

Neg sets z to $-x$ and returns z.

func (*Int) [Not](#)

```
func (z *Int) Not(x *Int) *Int
```

Not sets $z = \neg x$ and returns z.

func (*Int) [Or](#)

```
func (z *Int) Or(x, y *Int) *Int
```

Or sets $z = x | y$ and returns z .

func (*Int) [ProbablyPrime](#)

```
func (x *Int) ProbablyPrime(n int) bool
```

ProbablyPrime performs n Miller-Rabin tests to check whether x is prime. If it returns true, x is prime with probability $1 - 1/4^n$. If it returns false, x is not prime.

func (*Int) [Quo](#)

```
func (z *Int) Quo(x, y *Int) *Int
```

Quo sets z to the quotient x/y for $y \neq 0$ and returns z . If $y == 0$, a division-by-zero run-time panic occurs. Quo implements truncated division (like Go); see QuoRem for more details.

func (*Int) [QuoRem](#)

```
func (z *Int) QuoRem(x, y, r *Int) (*Int, *Int)
```

QuoRem sets z to the quotient x/y and r to the remainder $x\%y$ and returns the pair (z, r) for $y \neq 0$. If $y == 0$, a division-by-zero run-time panic occurs.

QuoRem implements T-division and modulus (like Go):

$$\begin{aligned} q &= x/y && \text{with the result truncated to zero} \\ r &= x - y*q \end{aligned}$$

(See Daan Leijen, “Division and Modulus for Computer Scientists”.) See DivMod for Euclidean division and modulus (unlike Go).

func (*Int) [Rand](#)

```
func (z *Int) Rand(rnd *rand.Rand, n *Int) *Int
```

Rand sets z to a pseudo-random number in [0, n) and returns z.

func (*Int) [Rem](#)

```
func (z *Int) Rem(x, y *Int) *Int
```

Rem sets z to the remainder $x\%y$ for $y \neq 0$ and returns z. If $y == 0$, a division-by-zero run-time panic occurs. Rem implements truncated modulus (like Go); see QuoRem for more details.

func (*Int) [Rsh](#)

```
func (z *Int) Rsh(x *Int, n uint) *Int
```

Rsh sets $z = x \gg n$ and returns z.

func (*Int) [Scan](#)

```
func (z *Int) Scan(s fmt.ScanState, ch rune) error
```

Scan is a support routine for `fmt.Scanner`; it sets z to the value of the scanned number. It accepts the formats 'b' (binary), 'o' (octal), 'd' (decimal), 'x' (lowercase hexadecimal), and 'X' (uppercase hexadecimal).

? Example

? Example

Code:

```
// The Scan function is rarely used directly;
// the fmt package recognizes it as an implementation of fmt.Scanner
i := new(big.Int)
_, err := fmt.Sscan("18446744073709551617", i)
if err != nil {
    log.Println("error scanning value:", err)
} else {
    fmt.Println(i)
}
```

Output:

18446744073709551617

func (*Int) [Set](#)

```
func (z *Int) Set(x *Int) *Int
```

Set sets z to x and returns z.

func (*Int) [SetBit](#)

```
func (z *Int) SetBit(x *Int, i int, b uint) *Int
```

SetBit sets z to x, with x's i'th bit set to b (0 or 1). That is, if bit is 1 SetBit sets $z = x | (1 \ll i)$; if bit is 0 it sets $z = x \& \sim (1 \ll i)$. If bit is not 0 or 1, SetBit will panic.

func (*Int) [SetBits](#)

```
func (z *Int) SetBits(abs []Word) *Int
```

SetBits provides raw (unchecked but fast) access to z by setting its value to abs, interpreted as a little-endian Word slice, and returning z. The result and abs share the same underlying array. SetBits is intended to support implementation of missing low-level Int functionality outside this package; it should be avoided otherwise.

func (*Int) [SetBytes](#)

```
func (z *Int) SetBytes(buf []byte) *Int
```

SetBytes interprets buf as the bytes of a big-endian unsigned integer, sets z to that value, and returns z.

func (*Int) [SetInt64](#)

```
func (z *Int) SetInt64(x int64) *Int
```

SetInt64 sets z to x and returns z.

func (*Int) [SetString](#)

```
func (z *Int) SetString(s string, base int) (*Int, bool)
```

SetString sets z to the value of s, interpreted in the given base, and returns z and a boolean indicating success. If SetString fails, the value of z is undefined but the returned value is nil.

The base argument must be 0 or a value from 2 through MaxBase. If the base is 0, the string prefix determines the actual conversion base. A prefix of “0x” or “0X” selects base 16; the “0” prefix selects base 8, and a “0b” or “0B” prefix selects base 2. Otherwise the selected base is 10.

? Example

? Example

Code:

```
i := new(big.Int)
i.SetString("644", 8) // octal
fmt.Println(i)
```

Output:

420

func (*Int) [Sign](#)

```
func (x *Int) Sign() int
```

Sign returns:

```
-1 if x < 0
 0 if x == 0
+1 if x > 0
```

func (*Int) [String](#)

```
func (x *Int) String() string
```

func (*Int) [Sub](#)

```
func (z *Int) Sub(x, y *Int) *Int
```

Sub sets z to the difference x-y and returns z.

func (*Int) [Xor](#)

```
func (z *Int) Xor(x, y *Int) *Int
```

Xor sets $z = x \wedge y$ and returns z.

type [Rat](#)

```
type Rat struct {  
    // contains filtered or unexported fields  
}
```

A Rat represents a quotient a/b of arbitrary precision. The zero value for a Rat represents the value 0.

func [NewRat](#)

```
func NewRat(a, b int64) *Rat
```

NewRat creates a new Rat with numerator a and denominator b.

func (*Rat) [Abs](#)

```
func (z *Rat) Abs(x *Rat) *Rat
```

Abs sets z to $|x|$ (the absolute value of x) and returns z.

func (*Rat) [Add](#)

```
func (z *Rat) Add(x, y *Rat) *Rat
```

Add sets z to the sum $x+y$ and returns z.

func (*Rat) [Cmp](#)

```
func (x *Rat) Cmp(y *Rat) int
```

Cmp compares x and y and returns:

```
-1 if x < y  
 0 if x == y  
+1 if x > y
```

func (*Rat) [Denom](#)

```
func (x *Rat) Denom() *Int
```

Denom returns the denominator of x; it is always > 0 . The result is a reference to x's denominator; it may change if a new value is assigned to x.

func (*Rat) [FloatString](#)

```
func (x *Rat) FloatString(prec int) string
```

FloatString returns a string representation of z in decimal form with prec digits of precision after the decimal point and the last digit rounded.

func (*Rat) [GobDecode](#)

```
func (z *Rat) GobDecode(buf []byte) error
```

GobDecode implements the gob.GobDecoder interface.

func (*Rat) [GobEncode](#)

```
func (x *Rat) GobEncode() ([]byte, error)
```

GobEncode implements the gob.GobEncoder interface.

func (*Rat) [Inv](#)

```
func (z *Rat) Inv(x *Rat) *Rat
```

Inv sets z to $1/x$ and returns z.

func (*Rat) [IsInt](#)

```
func (x *Rat) IsInt() bool
```

IsInt returns true if the denominator of x is 1.

func (*Rat) [Mul](#)

```
func (z *Rat) Mul(x, y *Rat) *Rat
```

Mul sets z to the product $x*y$ and returns z.

func (*Rat) [Neg](#)

```
func (z *Rat) Neg(x *Rat) *Rat
```

Neg sets z to -x and returns z.

func (*Rat) [Num](#)

```
func (x *Rat) Num() *Int
```

Num returns the numerator of x; it may be ≤ 0 . The result is a reference to x's numerator; it may change if a new value is assigned to x.

func (*Rat) [Quo](#)

```
func (z *Rat) Quo(x, y *Rat) *Rat
```

Quo sets z to the quotient x/y and returns z. If $y == 0$, a division-by-zero runtime panic occurs.

func (*Rat) [RatString](#)

```
func (x *Rat) RatString() string
```

RatString returns a string representation of z in the form "a/b" if $b \neq 1$, and in the form "a" if $b == 1$.

func (*Rat) [Scan](#)

```
func (z *Rat) Scan(s fmt.ScanState, ch rune) error
```

Scan is a support routine for `fmt.Scanner`. It accepts the formats 'e', 'E', 'f', 'F', 'g', 'G', and 'v'. All formats are equivalent.

? Example

? Example

Code:

```
// The Scan function is rarely used directly;
// the fmt package recognizes it as an implementation of fmt.Scanner
r := new(big.Rat)
_, err := fmt.Sscan("1.5000", r)
if err != nil {
    log.Println("error scanning value:", err)
} else {
    fmt.Println(r)
}
```

Output:

3/2

func (*Rat) [Set](#)

```
func (z *Rat) Set(x *Rat) *Rat
```

Set sets z to x (by making a copy of x) and returns z.

func (*Rat) [SetFrac](#)

```
func (z *Rat) SetFrac(a, b *Int) *Rat
```

SetFrac sets z to a/b and returns z.

func (*Rat) [SetFrac64](#)

```
func (z *Rat) SetFrac64(a, b int64) *Rat
```

SetFrac64 sets z to a/b and returns z.

func (*Rat) [SetInt](#)

```
func (z *Rat) SetInt(x *Int) *Rat
```

SetInt sets z to x (by making a copy of x) and returns z.

func (*Rat) [SetInt64](#)

```
func (z *Rat) SetInt64(x int64) *Rat
```

SetInt64 sets z to x and returns z.

func (*Rat) [SetString](#)

```
func (z *Rat) SetString(s string) (*Rat, bool)
```

SetString sets z to the value of s and returns z and a boolean indicating success. s can be given as a fraction "a/b" or as a floating-point number optionally followed by an exponent. If the operation failed, the value of z is undefined but the returned value is nil.

? Example

? Example

Code:

```
r := new(big.Rat)
r.SetString("355/113")
fmt.Println(r.FloatString(3))
```

Output:

3.142

func (*Rat) [Sign](#)

```
func (x *Rat) Sign() int
```

Sign returns:

```
-1 if x < 0
 0 if x == 0
+1 if x > 0
```

func (*Rat) [String](#)

```
func (x *Rat) String() string
```

String returns a string representation of z in the form "a/b" (even if b == 1).

func (*Rat) [Sub](#)

```
func (z *Rat) Sub(x, y *Rat) *Rat
```

Sub sets z to the difference x-y and returns z.

type [Word](#)

```
type word uintptr
```

A Word represents a single digit of a multi-precision unsigned integer.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package cmplx

```
import "math/cmplx"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `cmplx` provides basic constants and mathematical functions for complex numbers.

Index

[func Abs\(x complex128\) float64](#)
[func Acos\(x complex128\) complex128](#)
[func Acosh\(x complex128\) complex128](#)
[func Asin\(x complex128\) complex128](#)
[func Asinh\(x complex128\) complex128](#)
[func Atan\(x complex128\) complex128](#)
[func Atanh\(x complex128\) complex128](#)
[func Conj\(x complex128\) complex128](#)
[func Cos\(x complex128\) complex128](#)
[func Cosh\(x complex128\) complex128](#)
[func Cot\(x complex128\) complex128](#)
[func Exp\(x complex128\) complex128](#)
[func Inf\(\) complex128](#)
[func IsInf\(x complex128\) bool](#)
[func IsNaN\(x complex128\) bool](#)
[func Log\(x complex128\) complex128](#)
[func Log10\(x complex128\) complex128](#)
[func NaN\(\) complex128](#)
[func Phase\(x complex128\) float64](#)
[func Polar\(x complex128\) \(r, float64\)](#)
[func Pow\(x, y complex128\) complex128](#)
[func Rect\(r, float64\) complex128](#)
[func Sin\(x complex128\) complex128](#)
[func Sinh\(x complex128\) complex128](#)
[func Sqrt\(x complex128\) complex128](#)
[func Tan\(x complex128\) complex128](#)
[func Tanh\(x complex128\) complex128](#)

Package files

[abs.go](#) [asin.go](#) [conj.go](#) [exp.go](#) [isinf.go](#) [isnan.go](#) [log.go](#) [phase.go](#) [polar.go](#) [pow.go](#) [rect.go](#) [sin.go](#) [sqrt.go](#) [tan.go](#)

func [Abs](#)

```
func Abs(x complex128) float64
```

Abs returns the absolute value (also called the modulus) of x.

func [Acos](#)

```
func Acos(x complex128) complex128
```

Acos returns the inverse cosine of x.

func [Acosh](#)

`func Acosh(x complex128) complex128`

Acosh returns the inverse hyperbolic cosine of x.

func [Asin](#)

```
func Asin(x complex128) complex128
```

Asin returns the inverse sine of x.

func [Asinh](#)

```
func Asinh(x complex128) complex128
```

Asinh returns the inverse hyperbolic sine of x.

func [Atan](#)

```
func Atan(x complex128) complex128
```

Atan returns the inverse tangent of x.

func [Atanh](#)

`func Atanh(x complex128) complex128`

Atanh returns the inverse hyperbolic tangent of x.

func [Conj](#)

```
func Conj(x complex128) complex128
```

Conj returns the complex conjugate of x.

func Cos

```
func Cos(x complex128) complex128
```

Cos returns the cosine of x.

func Cosh

```
func Cosh(x complex128) complex128
```

Cosh returns the hyperbolic cosine of x .

func [Cot](#)

```
func Cot(x complex128) complex128
```

Cot returns the cotangent of x.

func [Exp](#)

```
func Exp(x complex128) complex128
```

Exp returns e^{**x} , the base-e exponential of x.

func [Inf](#)

```
func Inf() complex128
```

Inf returns a complex infinity, `complex(+Inf, +Inf)`.

func [IsInf](#)

```
func IsInf(x complex128) bool
```

IsInf returns true if either `real(x)` or `imag(x)` is an infinity.

func [IsNaN](#)

```
func IsNaN(x complex128) bool
```

IsNaN returns true if either `real(x)` or `imag(x)` is NaN and neither is an infinity.

func Log

```
func Log(x complex128) complex128
```

Log returns the natural logarithm of x.

func Log10

`func Log10(x complex128) complex128`

Log10 returns the decimal logarithm of x.

func [NaN](#)

```
func NaN() complex128
```

NaN returns a complex “not-a-number” value.

func [Phase](#)

```
func Phase(x complex128) float64
```

Phase returns the phase (also called the argument) of x . The returned value is in the range $[-\pi, \pi]$.

func [Polar](#)

```
func Polar(x complex128) (r, float64)
```

Polar returns the absolute value r and phase of x , such that $x = r * e^{**i}$. The phase is in the range $[-\text{Pi}, \text{Pi}]$.

func Pow

`func Pow(x, y complex128) complex128`

Pow returns $x^{**}y$, the base-x exponential of y.

func [Rect](#)

```
func Rect(r, float64) complex128
```

Rect returns the complex number x with polar coordinates r, θ .

func [Sin](#)

```
func Sin(x complex128) complex128
```

Sin returns the sine of x.

func [Sinh](#)

```
func Sinh(x complex128) complex128
```

Sinh returns the hyperbolic sine of x.

func [Sqrt](#)

```
func Sqrt(x complex128) complex128
```

Sqrt returns the square root of x.

func [Tan](#)

```
func Tan(x complex128) complex128
```

Tan returns the tangent of x.

func [Tanh](#)

```
func Tanh(x complex128) complex128
```

Tanh returns the hyperbolic tangent of x.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package rand

```
import "math/rand"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package rand implements pseudo-random number generators.

Index

[func ExpFloat64\(\) float64](#)

[func Float32\(\) float32](#)

[func Float64\(\) float64](#)

[func Int\(\) int](#)

[func Int31\(\) int32](#)

[func Int31n\(n int32\) int32](#)

[func Int63\(\) int64](#)

[func Int63n\(n int64\) int64](#)

[func Intn\(n int\) int](#)

[func NormFloat64\(\) float64](#)

[func Perm\(n int\) \[\]int](#)

[func Seed\(seed int64\)](#)

[func Uint32\(\) uint32](#)

[type Rand](#)

[func New\(src Source\) *Rand](#)

[func \(r *Rand\) ExpFloat64\(\) float64](#)

[func \(r *Rand\) Float32\(\) float32](#)

[func \(r *Rand\) Float64\(\) float64](#)

[func \(r *Rand\) Int\(\) int](#)

[func \(r *Rand\) Int31\(\) int32](#)

[func \(r *Rand\) Int31n\(n int32\) int32](#)

[func \(r *Rand\) Int63\(\) int64](#)

[func \(r *Rand\) Int63n\(n int64\) int64](#)

[func \(r *Rand\) Intn\(n int\) int](#)

[func \(r *Rand\) NormFloat64\(\) float64](#)

[func \(r *Rand\) Perm\(n int\) \[\]int](#)

[func \(r *Rand\) Seed\(seed int64\)](#)

[func \(r *Rand\) Uint32\(\) uint32](#)

[type Source](#)

[func NewSource\(seed int64\) Source](#)

[type Zipf](#)

[func NewZipf\(r *Rand, s float64, v float64, imax uint64\) *Zipf](#)

[func \(z *Zipf\) Uint64\(\) uint64](#)

Package files

[exp.go](#) [normal.go](#) [rand.go](#) [rng.go](#) [zipf.go](#)

func [ExpFloat64](#)

```
func ExpFloat64() float64
```

ExpFloat64 returns an exponentially distributed float64 in the range (0, +math.MaxFloat64] with an exponential distribution whose rate parameter (lambda) is 1 and whose mean is 1/lambda (1). To produce a distribution with a different rate parameter, callers can adjust the output using:

```
sample = ExpFloat64() / desiredRateParameter
```

func Float32

```
func Float32() float32
```

Float32 returns, as a float32, a pseudo-random number in [0.0,1.0).

func Float64

```
func Float64() float64
```

Float64 returns, as a float64, a pseudo-random number in [0.0,1.0).

func Int

```
func Int() int
```

Int returns a non-negative pseudo-random int.

func [Int31](#)

```
func Int31() int32
```

Int31 returns a non-negative pseudo-random 31-bit integer as an int32.

func [Int31n](#)

```
func Int31n(n int32) int32
```

Int31n returns, as an int32, a non-negative pseudo-random number in [0,n). It panics if $n \leq 0$.

func [Int63](#)

```
func Int63() int64
```

Int63 returns a non-negative pseudo-random 63-bit integer as an int64.

func Int63n

```
func Int63n(n int64) int64
```

Int63n returns, as an int64, a non-negative pseudo-random number in [0,n). It panics if $n \leq 0$.

func [Intn](#)

```
func Intn(n int) int
```

Intn returns, as an int, a non-negative pseudo-random number in [0,n). It panics if $n \leq 0$.

func [NormFloat64](#)

```
func NormFloat64() float64
```

NormFloat64 returns a normally distributed float64 in the range [-math.MaxFloat64, +math.MaxFloat64] with standard normal distribution (mean = 0, stddev = 1). To produce a different normal distribution, callers can adjust the output using:

```
sample = NormFloat64() * desiredStdDev + desiredMean
```

func Perm

```
func Perm(n int) []int
```

Perm returns, as a slice of n ints, a pseudo-random permutation of the integers [0,n).

func [Seed](#)

```
func Seed(seed int64)
```

Seed uses the provided seed value to initialize the generator to a deterministic state. If Seed is not called, the generator behaves as if seeded by Seed(1).

func [Uint32](#)

```
func Uint32() uint32
```

Uin32 returns a pseudo-random 32-bit value as a uint32.

type [Rand](#)

```
type Rand struct {  
    // contains filtered or unexported fields  
}
```

A Rand is a source of random numbers.

func [New](#)

```
func New(src Source) *Rand
```

New returns a new Rand that uses random values from src to generate other random values.

func (*Rand) [ExpFloat64](#)

```
func (r *Rand) ExpFloat64() float64
```

ExpFloat64 returns an exponentially distributed float64 in the range (0, +math.MaxFloat64] with an exponential distribution whose rate parameter (lambda) is 1 and whose mean is 1/lambda (1). To produce a distribution with a different rate parameter, callers can adjust the output using:

```
sample = ExpFloat64() / desiredRateParameter
```

func (*Rand) [Float32](#)

```
func (r *Rand) Float32() float32
```

Float32 returns, as a float32, a pseudo-random number in [0.0,1.0).

func (*Rand) [Float64](#)

```
func (r *Rand) Float64() float64
```

Float64 returns, as a float64, a pseudo-random number in [0.0,1.0).

func (*Rand) [Int](#)

```
func (r *Rand) Int() int
```

Int returns a non-negative pseudo-random int.

func (*Rand) [Int31](#)

```
func (r *Rand) Int31() int32
```

Int31 returns a non-negative pseudo-random 31-bit integer as an int32.

func (*Rand) [Int31n](#)

```
func (r *Rand) Int31n(n int32) int32
```

Int31n returns, as an int32, a non-negative pseudo-random number in [0,n). It panics if $n \leq 0$.

func (*Rand) [Int63](#)

```
func (r *Rand) Int63() int64
```

Int63 returns a non-negative pseudo-random 63-bit integer as an int64.

func (*Rand) [Int63n](#)

```
func (r *Rand) Int63n(n int64) int64
```

Int63n returns, as an int64, a non-negative pseudo-random number in [0,n). It panics if $n \leq 0$.

func (*Rand) [Intn](#)

```
func (r *Rand) Intn(n int) int
```

Intn returns, as an int, a non-negative pseudo-random number in [0,n). It panics if $n \leq 0$.

func (*Rand) [NormFloat64](#)

```
func (r *Rand) NormFloat64() float64
```

NormFloat64 returns a normally distributed float64 in the range [-math.MaxFloat64, +math.MaxFloat64] with standard normal distribution (mean = 0, stddev = 1). To produce a different normal distribution, callers can adjust the output using:

```
sample = NormFloat64() * desiredStdDev + desiredMean
```

func (*Rand) [Perm](#)

```
func (r *Rand) Perm(n int) []int
```

Perm returns, as a slice of n ints, a pseudo-random permutation of the integers [0,n).

func (*Rand) [Seed](#)

```
func (r *Rand) Seed(seed int64)
```

Seed uses the provided seed value to initialize the generator to a deterministic state.

func (*Rand) [Uint32](#)

```
func (r *Rand) Uint32() uint32
```

Uint32 returns a pseudo-random 32-bit value as a uint32.

type [Source](#)

```
type Source interface {  
    Int63() int64  
    Seed(seed int64)  
}
```

A Source represents a source of uniformly-distributed pseudo-random int64 values in the range $[0, 1 \ll 63)$.

func [NewSource](#)

```
func NewSource(seed int64) Source
```

NewSource returns a new pseudo-random Source seeded with the given value.

type [Zipf](#)

```
type Zipf struct {  
    // contains filtered or unexported fields  
}
```

A Zipf generates Zipf distributed variates.

func [NewZipf](#)

```
func NewZipf(r *Rand, s float64, v float64, imax uint64) *Zipf
```

NewZipf returns a Zipf generating variates $p(k)$ on $[0, imax]$ proportional to $(v+k)^{-s}$ where $s > 1$ and $k \geq 0$, and $v \geq 1$.

func (*Zipf) [Uint64](#)

```
func (z *Zipf) Uint64() uint64
```

Uint64 returns a value drawn from the Zipf distributed described by the Zipf object.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package mime

```
import "mime"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package mime implements parts of the MIME spec.

Index

[func AddExtensionType\(ext, typ string\) error](#)
[func FormatMediaType\(t string, param map\[string\]string\) string](#)
[func ParseMediaType\(v string\) \(mediatype string, params map\[string\]string, err error\)](#)
[func TypeByExtension\(ext string\) string](#)

Package files

[grammar.go](#) [mediatype.go](#) [type.go](#) [type_unix.go](#)

func [AddExtensionType](#)

```
func AddExtensionType(ext, typ string) error
```

AddExtensionType sets the MIME type associated with the extension ext to typ. The extension should begin with a leading dot, as in ".html".

func [FormatMediaType](#)

```
func FormatMediaType(t string, param map[string]string) string
```

FormatMediaType serializes mediatype t and the parameters param as a media type conforming to RFC 2045 and RFC 2616. The type and parameter names are written in lower-case. When any of the arguments result in a standard violation then FormatMediaType returns the empty string.

func [ParseMediaType](#)

```
func ParseMediaType(v string) (mediatype string, params map[string]s
```

ParseMediaType parses a media type value and any optional parameters, per RFC 1521. Media types are the values in Content-Type and Content-Disposition headers (RFC 2183). On success, ParseMediaType returns the media type converted to lowercase and trimmed of white space and a non-nil map. The returned map, params, maps from the lowercase attribute to the attribute value with its case preserved.

func [TypeByExtension](#)

```
func TypeByExtension(ext string) string
```

TypeByExtension returns the MIME type associated with the file extension ext. The extension ext should begin with a leading dot, as in ".html". When ext has no associated type, TypeByExtension returns "".

The built-in table is small but on unix it is augmented by the local system's mime.types file(s) if available under one or more of these names:

```
/etc/mime.types  
/etc/apache2/mime.types  
/etc/apache/mime.types
```

Windows system mime types are extracted from registry.

Text types have the charset parameter set to "utf-8" by default.

Subdirectories

Name	Synopsis
multipart	Package multipart implements MIME multipart parsing, as defined in RFC 2046.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package multipart

```
import "mime/multipart"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package multipart implements MIME multipart parsing, as defined in RFC 2046.

The implementation is sufficient for HTTP (RFC 2388) and the multipart bodies generated by popular browsers.

Index

[type File](#)

[type FileHeader](#)

[func \(fh *FileHeader\) Open\(\) \(File, error\)](#)

[type Form](#)

[func \(f *Form\) RemoveAll\(\) error](#)

[type Part](#)

[func \(p *Part\) Close\(\) error](#)

[func \(p *Part\) FileName\(\) string](#)

[func \(p *Part\) FormName\(\) string](#)

[func \(p *Part\) Read\(d \[\]byte\) \(n int, err error\)](#)

[type Reader](#)

[func NewReader\(reader io.Reader, boundary string\) *Reader](#)

[func \(r *Reader\) NextPart\(\) \(*Part, error\)](#)

[func \(r *Reader\) ReadForm\(maxMemory int64\) \(f *Form, err error\)](#)

[type Writer](#)

[func NewWriter\(w io.Writer\) *Writer](#)

[func \(w *Writer\) Boundary\(\) string](#)

[func \(w *Writer\) Close\(\) error](#)

[func \(w *Writer\) CreateFormField\(fieldname string\) \(io.Writer, error\)](#)

[func \(w *Writer\) CreateFormFile\(fieldname, filename string\) \(io.Writer,](#)

[error\)](#)

[func \(w *Writer\) CreatePart\(header textproto.MIMEHeader\) \(io.Writer,](#)

[error\)](#)

[func \(w *Writer\) FormDataContentType\(\) string](#)

[func \(w *Writer\) WriteField\(fieldname, value string\) error](#)

Package files

[formdata.go](#) [multipart.go](#) [writer.go](#)

type [File](#)

```
type File interface {  
    io.Reader  
    io.ReaderAt  
    io.Seeker  
    io.Closer  
}
```

File is an interface to access the file part of a multipart message. Its contents may be either stored in memory or on disk. If stored on disk, the File's underlying concrete type will be an `*os.File`.

type [FileHeader](#)

```
type FileHeader struct {  
    Filename string  
    Header    textproto.MIMEHeader  
    // contains filtered or unexported fields  
}
```

A FileHeader describes a file part of a multipart request.

func (*FileHeader) [Open](#)

```
func (fh *FileHeader) Open() (File, error)
```

Open opens and returns the FileHeader's associated File.

type [Form](#)

```
type Form struct {  
    Value map[string][]string  
    File  map[string][]*FileHeader  
}
```

Form is a parsed multipart form. Its File parts are stored either in memory or on disk, and are accessible via the *FileHeader's Open method. Its Value parts are stored as strings. Both are keyed by field name.

func (*Form) [RemoveAll](#)

```
func (f *Form) RemoveAll() error
```

RemoveAll removes any temporary files associated with a Form.

type [Part](#)

```
type Part struct {  
    // The headers of the body, if any, with the keys canonicalized  
    // in the same fashion that the Go http.Request headers are.  
    // i.e. "foo-bar" changes case to "Foo-Bar"  
    Header textproto.MIMEHeader  
    // contains filtered or unexported fields  
}
```

A Part represents a single part in a multipart body.

func (***Part**) [Close](#)

```
func (p *Part) Close() error
```

func (***Part**) [FileName](#)

```
func (p *Part) FileName() string
```

FileName returns the filename parameter of the Part's Content-Disposition header.

func (***Part**) [FormName](#)

```
func (p *Part) FormName() string
```

FormName returns the name parameter if p has a Content-Disposition of type "form-data". Otherwise it returns the empty string.

func (***Part**) [Read](#)

```
func (p *Part) Read(d []byte) (n int, err error)
```

Read reads the body of a part, after its headers and before the next part (if any) begins.

type [Reader](#)

```
type Reader struct {  
    // contains filtered or unexported fields  
}
```

Reader is an iterator over parts in a MIME multipart body. Reader's underlying parser consumes its input as needed. Seeking isn't supported.

func [NewReader](#)

```
func NewReader(reader io.Reader, boundary string) *Reader
```

NewReader creates a new multipart Reader reading from r using the given MIME boundary.

func (***Reader**) [NextPart](#)

```
func (r *Reader) NextPart() (*Part, error)
```

NextPart returns the next part in the multipart or an error. When there are no more parts, the error io.EOF is returned.

func (***Reader**) [ReadForm](#)

```
func (r *Reader) ReadForm(maxMemory int64) (f *Form, err error)
```

ReadForm parses an entire multipart message whose parts have a Content-Disposition of "form-data". It stores up to maxMemory bytes of the file parts in memory and the remainder on disk in temporary files.

type [Writer](#)

```
type Writer struct {  
    // contains filtered or unexported fields  
}
```

A Writer generates multipart messages.

func [NewWriter](#)

```
func NewWriter(w io.Writer) *Writer
```

NewWriter returns a new multipart Writer with a random boundary, writing to w.

func (***Writer**) [Boundary](#)

```
func (w *Writer) Boundary() string
```

Boundary returns the Writer's randomly selected boundary string.

func (***Writer**) [Close](#)

```
func (w *Writer) Close() error
```

Close finishes the multipart message and writes the trailing boundary end line to the output.

func (***Writer**) [CreateFormField](#)

```
func (w *Writer) CreateFormField(fieldname string) (io.Writer, error)
```

CreateFormField calls CreatePart with a header using the given field name.

func (***Writer**) [CreateFormFile](#)

```
func (w *Writer) CreateFormFile(fieldname, filename string) (io.Writ
```

CreateFormFile is a convenience wrapper around CreatePart. It creates a new form-data header with the provided field name and file name.

func (*Writer) [CreatePart](#)

```
func (w *Writer) CreatePart(header textproto.MIMEHeader) (io.Writer,
```

CreatePart creates a new multipart section with the provided header. The body of the part should be written to the returned Writer. After calling CreatePart, any previous part may no longer be written to.

func (*Writer) [FormDataContentType](#)

```
func (w *Writer) FormDataContentType() string
```

FormDataContentType returns the Content-Type for an HTTP multipart/form-data with this Writer's Boundary.

func (*Writer) [WriteField](#)

```
func (w *Writer) WriteField(fieldname, value string) error
```

WriteField calls CreateFormField and then writes the given value.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package net

```
import "net"
```

[Overview](#)

[Index](#)

[Examples](#)

[Subdirectories](#)

Overview ?

Overview ?

Package net provides a portable interface for network I/O, including TCP/IP, UDP, domain name resolution, and Unix domain sockets.

Although the package provides access to low-level networking primitives, most clients will need only the basic interface provided by the Dial, Listen, and Accept functions and the associated Conn and Listener interfaces. The crypto/tls package uses the same interfaces and similar Dial and Listen functions.

The Dial function connects to a server:

```
conn, err := net.Dial("tcp", "google.com:80")
if err != nil {
    // handle error
}
fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
status, err := bufio.NewReader(conn).ReadString('\n')
// ...
```

The Listen function creates servers:

```
ln, err := net.Listen("tcp", ":8080")
if err != nil {
    // handle error
}
for {
    conn, err := ln.Accept()
    if err != nil {
        // handle error
        continue
    }
    go handleConnection(conn)
}
```

Index

Constants

Variables

[func InterfaceAddrs\(\) \(\[\]Addr, error\)](#)

[func Interfaces\(\) \(\[\]Interface, error\)](#)

[func JoinHostPort\(host, port string\) string](#)

[func LookupAddr\(addr string\) \(name \[\]string, err error\)](#)

[func LookupCNAME\(name string\) \(cname string, err error\)](#)

[func LookupHost\(host string\) \(addrs \[\]string, err error\)](#)

[func LookupIP\(host string\) \(addrs \[\]IP, err error\)](#)

[func LookupMX\(name string\) \(mx \[\]*MX, err error\)](#)

[func LookupPort\(network, service string\) \(port int, err error\)](#)

[func LookupSRV\(service, proto, name string\) \(cname string, addrs \[\]*SRV, err error\)](#)

[func LookupTXT\(name string\) \(txt \[\]string, err error\)](#)

[func SplitHostPort\(hostport string\) \(host, port string, err error\)](#)

[type Addr](#)

[type AddrError](#)

[func \(e *AddrError\) Error\(\) string](#)

[func \(e *AddrError\) Temporary\(\) bool](#)

[func \(e *AddrError\) Timeout\(\) bool](#)

[type Conn](#)

[func Dial\(net, addr string\) \(Conn, error\)](#)

[func DialTimeout\(net, addr string, timeout time.Duration\) \(Conn, error\)](#)

[func FileConn\(f *os.File\) \(c Conn, err error\)](#)

[func Pipe\(\) \(Conn, Conn\)](#)

[type DNSConfigError](#)

[func \(e *DNSConfigError\) Error\(\) string](#)

[func \(e *DNSConfigError\) Temporary\(\) bool](#)

[func \(e *DNSConfigError\) Timeout\(\) bool](#)

[type DNSError](#)

[func \(e *DNSError\) Error\(\) string](#)

[func \(e *DNSError\) Temporary\(\) bool](#)

[func \(e *DNSError\) Timeout\(\) bool](#)

[type Error](#)

[type Flags](#)

[func \(f Flags\) String\(\) string](#)
[type HardwareAddr](#)
[func ParseMAC\(s string\) \(hw HardwareAddr, err error\)](#)
[func \(a HardwareAddr\) String\(\) string](#)
[type IP](#)
[func IPv4\(a, b, c, d byte\) IP](#)
[func ParseCIDR\(s string\) \(IP, *IPNet, error\)](#)
[func ParseIP\(s string\) IP](#)
[func \(ip IP\) DefaultMask\(\) IPMask](#)
[func \(ip IP\) Equal\(x IP\) bool](#)
[func \(ip IP\) IsGlobalUnicast\(\) bool](#)
[func \(ip IP\) IsInterfaceLocalMulticast\(\) bool](#)
[func \(ip IP\) IsLinkLocalMulticast\(\) bool](#)
[func \(ip IP\) IsLinkLocalUnicast\(\) bool](#)
[func \(ip IP\) IsLoopback\(\) bool](#)
[func \(ip IP\) IsMulticast\(\) bool](#)
[func \(ip IP\) IsUnspecified\(\) bool](#)
[func \(ip IP\) Mask\(mask IPMask\) IP](#)
[func \(ip IP\) String\(\) string](#)
[func \(ip IP\) To16\(\) IP](#)
[func \(ip IP\) To4\(\) IP](#)
[type IPAddr](#)
[func ResolveIPAddr\(net, addr string\) \(*IPAddr, error\)](#)
[func \(a *IPAddr\) Network\(\) string](#)
[func \(a *IPAddr\) String\(\) string](#)
[type IPConn](#)
[func DialIP\(netProto string, laddr, raddr *IPAddr\) \(*IPConn, error\)](#)
[func ListenIP\(netProto string, laddr *IPAddr\) \(*IPConn, error\)](#)
[func \(c *IPConn\) Close\(\) error](#)
[func \(c *IPConn\) File\(\) \(f *os.File, err error\)](#)
[func \(c *IPConn\) LocalAddr\(\) Addr](#)
[func \(c *IPConn\) Read\(b \[\]byte\) \(int, error\)](#)
[func \(c *IPConn\) ReadFrom\(b \[\]byte\) \(int, Addr, error\)](#)
[func \(c *IPConn\) ReadFromIP\(b \[\]byte\) \(int, *IPAddr, error\)](#)
[func \(c *IPConn\) RemoteAddr\(\) Addr](#)
[func \(c *IPConn\) SetDeadline\(t time.Time\) error](#)
[func \(c *IPConn\) SetReadBuffer\(bytes int\) error](#)
[func \(c *IPConn\) SetReadDeadline\(t time.Time\) error](#)
[func \(c *IPConn\) SetWriteBuffer\(bytes int\) error](#)

func (c *IPConn) SetWriteDeadline(t time.Time) error
func (c *IPConn) Write(b []byte) (int, error)
func (c *IPConn) WriteTo(b []byte, addr Addr) (int, error)
func (c *IPConn) WriteToIP(b []byte, addr *IPAddr) (int, error)

type IPMask

func CIDRMask(ones, bits int) IPMask
func IPv4Mask(a, b, c, d byte) IPMask
func (m IPMask) Size() (ones, bits int)
func (m IPMask) String() string

type IPNet

func (n *IPNet) Contains(ip IP) bool
func (n *IPNet) Network() string
func (n *IPNet) String() string

type Interface

func InterfaceByIndex(index int) (*Interface, error)
func InterfaceByName(name string) (*Interface, error)
func (ifi *Interface) Addrs() ([]Addr, error)
func (ifi *Interface) MulticastAddrs() ([]Addr, error)

type InvalidAddrError

func (e InvalidAddrError) Error() string
func (e InvalidAddrError) Temporary() bool
func (e InvalidAddrError) Timeout() bool

type Listener

func FileListener(f *os.File) (l Listener, err error)
func Listen(net, laddr string) (Listener, error)

type MX

type OpError

func (e *OpError) Error() string
func (e *OpError) Temporary() bool
func (e *OpError) Timeout() bool

type PacketConn

func FilePacketConn(f *os.File) (c PacketConn, err error)
func ListenPacket(net, addr string) (PacketConn, error)

type ParseError

func (e *ParseError) Error() string

type SRV

type TCPAddr

func ResolveTCPAddr(net, addr string) (*TCPAddr, error)
func (a *TCPAddr) Network() string

[func \(a *TCPAddr\) String\(\) string](#)

[type TCPConn](#)

[func DialTCP\(net string, laddr, raddr *TCPAddr\) \(*TCPConn, error\)](#)

[func \(c *TCPConn\) Close\(\) error](#)

[func \(c *TCPConn\) CloseRead\(\) error](#)

[func \(c *TCPConn\) CloseWrite\(\) error](#)

[func \(c *TCPConn\) File\(\) \(f *os.File, err error\)](#)

[func \(c *TCPConn\) LocalAddr\(\) Addr](#)

[func \(c *TCPConn\) Read\(b \[\]byte\) \(n int, err error\)](#)

[func \(c *TCPConn\) ReadFrom\(r io.Reader\) \(int64, error\)](#)

[func \(c *TCPConn\) RemoteAddr\(\) Addr](#)

[func \(c *TCPConn\) SetDeadline\(t time.Time\) error](#)

[func \(c *TCPConn\) SetKeepAlive\(keepalive bool\) error](#)

[func \(c *TCPConn\) SetLinger\(sec int\) error](#)

[func \(c *TCPConn\) SetNoDelay\(noDelay bool\) error](#)

[func \(c *TCPConn\) SetReadBuffer\(bytes int\) error](#)

[func \(c *TCPConn\) SetReadDeadline\(t time.Time\) error](#)

[func \(c *TCPConn\) SetWriteBuffer\(bytes int\) error](#)

[func \(c *TCPConn\) SetWriteDeadline\(t time.Time\) error](#)

[func \(c *TCPConn\) Write\(b \[\]byte\) \(n int, err error\)](#)

[type TCPListener](#)

[func ListenTCP\(net string, laddr *TCPAddr\) \(*TCPListener, error\)](#)

[func \(l *TCPListener\) Accept\(\) \(c Conn, err error\)](#)

[func \(l *TCPListener\) AcceptTCP\(\) \(c *TCPConn, err error\)](#)

[func \(l *TCPListener\) Addr\(\) Addr](#)

[func \(l *TCPListener\) Close\(\) error](#)

[func \(l *TCPListener\) File\(\) \(f *os.File, err error\)](#)

[func \(l *TCPListener\) SetDeadline\(t time.Time\) error](#)

[type UDPAddr](#)

[func ResolveUDPAddr\(net, addr string\) \(*UDPAddr, error\)](#)

[func \(a *UDPAddr\) Network\(\) string](#)

[func \(a *UDPAddr\) String\(\) string](#)

[type UDPConn](#)

[func DialUDP\(net string, laddr, raddr *UDPAddr\) \(*UDPConn, error\)](#)

[func ListenMulticastUDP\(net string, ifi *Interface, gaddr *UDPAddr\) \(*UDPConn, error\)](#)

[func ListenUDP\(net string, laddr *UDPAddr\) \(*UDPConn, error\)](#)

[func ListenUnixgram\(net string, laddr *UnixAddr\) \(*UDPConn, error\)](#)

[func \(c *UDPConn\) Close\(\) error](#)

[func \(c *UDPConn\) File\(\) \(f *os.File, err error\)](#)
[func \(c *UDPConn\) LocalAddr\(\) Addr](#)
[func \(c *UDPConn\) Read\(b \[\]byte\) \(int, error\)](#)
[func \(c *UDPConn\) ReadFrom\(b \[\]byte\) \(int, Addr, error\)](#)
[func \(c *UDPConn\) ReadFromUDP\(b \[\]byte\) \(n int, addr *UDPAddr, err error\)](#)
[func \(c *UDPConn\) RemoteAddr\(\) Addr](#)
[func \(c *UDPConn\) SetDeadline\(t time.Time\) error](#)
[func \(c *UDPConn\) SetReadBuffer\(bytes int\) error](#)
[func \(c *UDPConn\) SetReadDeadline\(t time.Time\) error](#)
[func \(c *UDPConn\) SetWriteBuffer\(bytes int\) error](#)
[func \(c *UDPConn\) SetWriteDeadline\(t time.Time\) error](#)
[func \(c *UDPConn\) Write\(b \[\]byte\) \(int, error\)](#)
[func \(c *UDPConn\) WriteTo\(b \[\]byte, addr Addr\) \(int, error\)](#)
[func \(c *UDPConn\) WriteToUDP\(b \[\]byte, addr *UDPAddr\) \(int, error\)](#)

[type UnixAddr](#)
[func ResolveUnixAddr\(net, addr string\) \(*UnixAddr, error\)](#)
[func \(a *UnixAddr\) Network\(\) string](#)
[func \(a *UnixAddr\) String\(\) string](#)

[type UnixConn](#)
[func DialUnix\(net string, laddr, raddr *UnixAddr\) \(*UnixConn, error\)](#)
[func \(c *UnixConn\) Close\(\) error](#)
[func \(c *UnixConn\) File\(\) \(f *os.File, err error\)](#)
[func \(c *UnixConn\) LocalAddr\(\) Addr](#)
[func \(c *UnixConn\) Read\(b \[\]byte\) \(n int, err error\)](#)
[func \(c *UnixConn\) ReadFrom\(b \[\]byte\) \(n int, addr Addr, err error\)](#)
[func \(c *UnixConn\) ReadFromUnix\(b \[\]byte\) \(n int, addr *UnixAddr, err error\)](#)
[func \(c *UnixConn\) ReadMsgUnix\(b, oob \[\]byte\) \(n, oobn, flags int, addr *UnixAddr, err error\)](#)
[func \(c *UnixConn\) RemoteAddr\(\) Addr](#)
[func \(c *UnixConn\) SetDeadline\(t time.Time\) error](#)
[func \(c *UnixConn\) SetReadBuffer\(bytes int\) error](#)
[func \(c *UnixConn\) SetReadDeadline\(t time.Time\) error](#)
[func \(c *UnixConn\) SetWriteBuffer\(bytes int\) error](#)
[func \(c *UnixConn\) SetWriteDeadline\(t time.Time\) error](#)
[func \(c *UnixConn\) Write\(b \[\]byte\) \(n int, err error\)](#)
[func \(c *UnixConn\) WriteMsgUnix\(b, oob \[\]byte, addr *UnixAddr\) \(n, oobn int, err error\)](#)

[func \(c *UnixConn\) WriteTo\(b \[\]byte, addr Addr\) \(n int, err error\)](#)
[func \(c *UnixConn\) WriteToUnix\(b \[\]byte, addr *UnixAddr\) \(n int, err error\)](#)
[type UnixListener](#)
[func ListenUnix\(net string, laddr *UnixAddr\) \(*UnixListener, error\)](#)
[func \(l *UnixListener\) Accept\(\) \(c Conn, err error\)](#)
[func \(l *UnixListener\) AcceptUnix\(\) \(*UnixConn, error\)](#)
[func \(l *UnixListener\) Addr\(\) Addr](#)
[func \(l *UnixListener\) Close\(\) error](#)
[func \(l *UnixListener\) File\(\) \(f *os.File, err error\)](#)
[func \(l *UnixListener\) SetDeadline\(t time.Time\) \(err error\)](#)
[type UnknownNetworkError](#)
[func \(e UnknownNetworkError\) Error\(\) string](#)
[func \(e UnknownNetworkError\) Temporary\(\) bool](#)
[func \(e UnknownNetworkError\) Timeout\(\) bool](#)
[Bugs](#)

Examples

[Listener](#)

Package files

[cgo](#) [linux.go](#) [cgo_unix.go](#) [dial.go](#) [dnsclient.go](#) [dnsclient_unix.go](#) [dnsconfig.go](#) [dnsmsg.go](#) [doc.go](#) [fd.go](#) [fd_linux.go](#) [file.go](#) [hosts.go](#) [interface.go](#) [interface_linux.go](#) [ip.go](#) [iprawsock.go](#) [iprawsock_posix.go](#) [ipsock.go](#) [ipsock_posix.go](#) [lookup_unix.go](#) [mac.go](#) [net.go](#) [newpollserver.go](#) [parse.go](#) [pipe.go](#) [port.go](#) [sendfile_linux.go](#) [sock.go](#) [sock_linux.go](#) [sockopt.go](#) [sockopt_linux.go](#) [sockoptip.go](#) [sockoptip_linux.go](#) [tcpsock.go](#) [tcpsock_posix.go](#) [udpsock.go](#) [udpsock_posix.go](#) [unixsock.go](#) [unixsock_posix.go](#)

Constants

```
const (  
    IPv4len = 4  
    IPv6len = 16  
)
```

IP address lengths (bytes).

Variables

```
var (  
    IPv4bcast      = IPv4(255, 255, 255, 255) // broadcast  
    IPv4allsys    = IPv4(224, 0, 0, 1)       // all systems  
    IPv4allrouter = IPv4(224, 0, 0, 2)       // all routers  
    IPv4zero      = IPv4(0, 0, 0, 0)        // all zeros  
)
```

Well-known IPv4 addresses

```
var (  
    IPv6zero          = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
    IPv6unspecified  = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
    IPv6loopback     = IP{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
    IPv6interfacelocalallnodes = IP{0xff, 0x01, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
    IPv6linklocalallnodes    = IP{0xff, 0x02, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
    IPv6linklocalallrouters  = IP{0xff, 0x02, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
)
```

Well-known IPv6 addresses

```
var ErrWriteToConnected = errors.New("use of WriteTo with pre-connec
```

func [InterfaceAddr](#)

```
func InterfaceAddr() ([]Addr, error)
```

InterfaceAddr returns a list of the system's network interface addresses.

func [Interfaces](#)

```
func Interfaces() ([]Interface, error)
```

Interfaces returns a list of the system's network interfaces.

func [JoinHostPort](#)

```
func JoinHostPort(host, port string) string
```

JoinHostPort combines host and port into a network address of the form "host:port" or, if host contains a colon, "[host]:port".

func LookupAddr

```
func LookupAddr(addr string) (name []string, err error)
```

LookupAddr performs a reverse lookup for the given address, returning a list of names mapping to that address.

func LookupCNAME

```
func LookupCNAME(name string) (cname string, err error)
```

LookupCNAME returns the canonical DNS host for the given name. Callers that do not care about the canonical name can call LookupHost or LookupIP directly; both take care of resolving the canonical name as part of the lookup.

func LookupHost

```
func LookupHost(host string) (addrs []string, err error)
```

LookupHost looks up the given host using the local resolver. It returns an array of that host's addresses.

func [LookupIP](#)

```
func LookupIP(host string) (addrs []IP, err error)
```

LookupIP looks up host using the local resolver. It returns an array of that host's IPv4 and IPv6 addresses.

func LookupMX

```
func LookupMX(name string) (mx []*MX, err error)
```

LookupMX returns the DNS MX records for the given domain name sorted by preference.

func LookupPort

```
func LookupPort(network, service string) (port int, err error)
```

LookupPort looks up the port for the given network and service.

func [LookupSRV](#)

```
func LookupSRV(service, proto, name string) (cname string, addrs []*)
```

LookupSRV tries to resolve an SRV query of the given service, protocol, and domain name. The proto is "tcp" or "udp". The returned records are sorted by priority and randomized by weight within a priority.

LookupSRV constructs the DNS name to look up following RFC 2782. That is, it looks up `_service._proto.name`. To accommodate services publishing SRV records under non-standard names, if both service and proto are empty strings, LookupSRV looks up name directly.

func LookupTXT

```
func LookupTXT(name string) (txt []string, err error)
```

LookupTXT returns the DNS TXT records for the given domain name.

func [SplitHostPort](#)

```
func SplitHostPort(hostport string) (host, port string, err error)
```

SplitHostPort splits a network address of the form "host:port" or "[host]:port" into host and port. The latter form must be used when host contains a colon.

type [Addr](#)

```
type Addr interface {  
    Network() string // name of the network  
    String() string  // string form of address  
}
```

Addr represents a network end point address.

type [AddrError](#)

```
type AddrError struct {  
    Err string  
    Addr string  
}
```

func (*AddrError) [Error](#)

```
func (e *AddrError) Error() string
```

func (*AddrError) [Temporary](#)

```
func (e *AddrError) Temporary() bool
```

func (*AddrError) [Timeout](#)

```
func (e *AddrError) Timeout() bool
```

type [Conn](#)

```
type Conn interface {
    // Read reads data from the connection.
    // Read can be made to time out and return a Error with Timeout(
    // after a fixed time limit; see SetDeadline and SetReadDeadline
    Read(b []byte) (n int, err error)

    // Write writes data to the connection.
    // Write can be made to time out and return a Error with Timeout
    // after a fixed time limit; see SetDeadline and SetWriteDeadlin
    Write(b []byte) (n int, err error)

    // Close closes the connection.
    // Any blocked Read or Write operations will be unblocked and re
    Close() error

    // LocalAddr returns the local network address.
    LocalAddr() Addr

    // RemoteAddr returns the remote network address.
    RemoteAddr() Addr

    // SetDeadline sets the read and write deadlines associated
    // with the connection. It is equivalent to calling both
    // SetReadDeadline and SetWriteDeadline.
    //
    // A deadline is an absolute time after which I/O operations
    // fail with a timeout (see type Error) instead of
    // blocking. The deadline applies to all future I/O, not just
    // the immediately following call to Read or Write.
    //
    // An idle timeout can be implemented by repeatedly extending
    // the deadline after successful Read or Write calls.
    //
    // A zero value for t means I/O operations will not time out.
    SetDeadline(t time.Time) error

    // SetReadDeadline sets the deadline for future Read calls.
    // A zero value for t means Read will not time out.
    SetReadDeadline(t time.Time) error

    // SetWriteDeadline sets the deadline for future Write calls.
    // Even if write times out, it may return n > 0, indicating that
    // some of the data was successfully written.
    // A zero value for t means Write will not time out.
    SetWriteDeadline(t time.Time) error
}
```

```
}
```

Conn is a generic stream-oriented network connection.

Multiple goroutines may invoke methods on a Conn simultaneously.

func [Dial](#)

```
func Dial(net, addr string) (Conn, error)
```

Dial connects to the address addr on the network net.

Known networks are "tcp", "tcp4" (IPv4-only), "tcp6" (IPv6-only), "udp", "udp4" (IPv4-only), "udp6" (IPv6-only), "ip", "ip4" (IPv4-only), "ip6" (IPv6-only), "unix" and "unixpacket".

For TCP and UDP networks, addresses have the form host:port. If host is a literal IPv6 address, it must be enclosed in square brackets. The functions JoinHostPort and SplitHostPort manipulate addresses in this form.

Examples:

```
Dial("tcp", "12.34.56.78:80")
Dial("tcp", "google.com:80")
Dial("tcp", "[de:ad:be:ef::ca:fe]:80")
```

For IP networks, addr must be "ip", "ip4" or "ip6" followed by a colon and a protocol number or name.

Examples:

```
Dial("ip4:1", "127.0.0.1")
Dial("ip6:ospf", ":::1")
```

func [DialTimeout](#)

```
func DialTimeout(net, addr string, timeout time.Duration) (Conn, err
```

DialTimeout acts like Dial but takes a timeout. The timeout includes name resolution, if required.

func [FileConn](#)

```
func FileConn(f *os.File) (c Conn, err error)
```

FileConn returns a copy of the network connection corresponding to the open file `f`. It is the caller's responsibility to close `f` when finished. Closing `c` does not affect `f`, and closing `f` does not affect `c`.

func [Pipe](#)

```
func Pipe() (Conn, Conn)
```

Pipe creates a synchronous, in-memory, full duplex network connection; both ends implement the Conn interface. Reads on one end are matched with writes on the other, copying data directly between the two; there is no internal buffering.

type [DNSConfigError](#)

```
type DNSConfigError struct {  
    Err error  
}
```

DNSConfigError represents an error reading the machine's DNS configuration.

func (*DNSConfigError) [Error](#)

```
func (e *DNSConfigError) Error() string
```

func (*DNSConfigError) [Temporary](#)

```
func (e *DNSConfigError) Temporary() bool
```

func (*DNSConfigError) [Timeout](#)

```
func (e *DNSConfigError) Timeout() bool
```

type [DNSError](#)

```
type DNSError struct {  
    Err      string // description of the error  
    Name     string // name looked for  
    Server   string // server used  
    IsTimeout bool  
}
```

DNSError represents a DNS lookup error.

func (***DNSError**) [Error](#)

```
func (e *DNSError) Error() string
```

func (***DNSError**) [Temporary](#)

```
func (e *DNSError) Temporary() bool
```

func (***DNSError**) [Timeout](#)

```
func (e *DNSError) Timeout() bool
```

type [Error](#)

```
type Error interface {  
    error  
    Timeout() bool // Is the error a timeout?  
    Temporary() bool // Is the error temporary?  
}
```

An Error represents a network error.

type [Flags](#)

```
type Flags uint
```

```
const (  
    FlagUp           Flags = 1 << iota // interface is up  
    FlagBroadcast    // interface supports broadca  
    FlagLoopback     // interface is a loopback in  
    FlagPointToPoint // interface belongs to a poi  
    FlagMulticast    // interface supports multica  
)
```

func (Flags) [String](#)

```
func (f Flags) String() string
```

type [HardwareAddr](#)

```
type HardwareAddr []byte
```

A HardwareAddr represents a physical hardware address.

func [ParseMAC](#)

```
func ParseMAC(s string) (hw HardwareAddr, err error)
```

ParseMAC parses s as an IEEE 802 MAC-48, EUI-48, or EUI-64 using one of the following formats:

```
01:23:45:67:89:ab  
01:23:45:67:89:ab:cd:ef  
01-23-45-67-89-ab  
01-23-45-67-89-ab-cd-ef  
0123.4567.89ab  
0123.4567.89ab.cdef
```

func (HardwareAddr) [String](#)

```
func (a HardwareAddr) String() string
```

type [IP](#)

type IP []byte

An IP is a single IP address, an array of bytes. Functions in this package accept either 4-byte (IPv4) or 16-byte (IPv6) arrays as input.

Note that in this documentation, referring to an IP address as an IPv4 address or an IPv6 address is a semantic property of the address, not just the length of the byte array: a 16-byte array can still be an IPv4 address.

func [IPv4](#)

func IPv4(a, b, c, d byte) IP

IPv4 returns the IP address (in 16-byte form) of the IPv4 address a.b.c.d.

func [ParseCIDR](#)

func ParseCIDR(s string) (IP, *IPNet, error)

ParseCIDR parses s as a CIDR notation IP address and mask, like "192.168.100.1/24" or "2001:DB8::/48", as defined in RFC 4632 and RFC 4291.

It returns the IP address and the network implied by the IP and mask. For example, ParseCIDR("192.168.100.1/16") returns the IP address 192.168.100.1 and the network 192.168.0.0/16.

func [ParseIP](#)

func ParseIP(s string) IP

ParseIP parses s as an IP address, returning the result. The string s can be in dotted decimal ("74.125.19.99") or IPv6 ("2001:4860:0:2001::68") form. If s is not a valid textual representation of an IP address, ParseIP returns nil.

func (IP) [DefaultMask](#)

```
func (ip IP) DefaultMask() IPMask
```

DefaultMask returns the default IP mask for the IP address ip. Only IPv4 addresses have default masks; DefaultMask returns nil if ip is not a valid IPv4 address.

func (IP) [Equal](#)

```
func (ip IP) Equal(x IP) bool
```

Equal returns true if ip and x are the same IP address. An IPv4 address and that same address in IPv6 form are considered to be equal.

func (IP) [IsGlobalUnicast](#)

```
func (ip IP) IsGlobalUnicast() bool
```

IsGlobalUnicast returns true if ip is a global unicast address.

func (IP) [IsInterfaceLocalMulticast](#)

```
func (ip IP) IsInterfaceLocalMulticast() bool
```

IsInterfaceLinkLocalMulticast returns true if ip is an interface-local multicast address.

func (IP) [IsLinkLocalMulticast](#)

```
func (ip IP) IsLinkLocalMulticast() bool
```

IsLinkLocalMulticast returns true if ip is a link-local multicast address.

func (IP) [IsLinkLocalUnicast](#)

```
func (ip IP) IsLinkLocalUnicast() bool
```

IsLinkLocalUnicast returns true if ip is a link-local unicast address.

func (IP) [IsLoopback](#)

```
func (ip IP) IsLoopback() bool
```

IsLoopback returns true if ip is a loopback address.

func (IP) [IsMulticast](#)

```
func (ip IP) IsMulticast() bool
```

IsMulticast returns true if ip is a multicast address.

func (IP) [IsUnspecified](#)

```
func (ip IP) IsUnspecified() bool
```

IsUnspecified returns true if ip is an unspecified address.

func (IP) [Mask](#)

```
func (ip IP) Mask(mask IPMask) IP
```

Mask returns the result of masking the IP address ip with mask.

func (IP) [String](#)

```
func (ip IP) String() string
```

String returns the string form of the IP address ip. If the address is an IPv4 address, the string representation is dotted decimal ("74.125.19.99"). Otherwise the representation is IPv6 ("2001:4860:0:2001::68").

func (IP) [To16](#)

```
func (ip IP) To16() IP
```

To16 converts the IP address ip to a 16-byte representation. If ip is not an IP address (it is the wrong length), To16 returns nil.

func (IP) [To4](#)

```
func (ip IP) To4() IP
```

To4 converts the IPv4 address ip to a 4-byte representation. If ip is not an IPv4 address, To4 returns nil.

type [IPAddr](#)

```
type IPAddr struct {  
    IP IP  
}
```

IPAddr represents the address of a IP end point.

func [ResolveIPAddr](#)

```
func ResolveIPAddr(net, addr string) (*IPAddr, error)
```

ResolveIPAddr parses addr as a IP address and resolves domain names to numeric addresses on the network net, which must be "ip", "ip4" or "ip6". A literal IPv6 host address must be enclosed in square brackets, as in "[::]".

func (***IPAddr**) [Network](#)

```
func (a *IPAddr) Network() string
```

Network returns the address's network name, "ip".

func (***IPAddr**) [String](#)

```
func (a *IPAddr) String() string
```

type [IPConn](#)

```
type IPConn struct {  
    // contains filtered or unexported fields  
}
```

IPConn is the implementation of the Conn and PacketConn interfaces for IP network connections.

func [DialIP](#)

```
func DialIP(netProto string, laddr, raddr *IPAddr) (*IPConn, error)
```

DialIP connects to the remote address raddr on the network protocol netProto, which must be "ip", "ip4", or "ip6" followed by a colon and a protocol number or name.

func [ListenIP](#)

```
func ListenIP(netProto string, laddr *IPAddr) (*IPConn, error)
```

ListenIP listens for incoming IP packets addressed to the local address laddr. The returned connection c's ReadFrom and WriteTo methods can be used to receive and send IP packets with per-packet addressing.

func (***IPConn**) [Close](#)

```
func (c *IPConn) Close() error
```

Close closes the IP connection.

func (***IPConn**) [File](#)

```
func (c *IPConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File, set to blocking mode. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

func (*IPConn) [LocalAddr](#)

```
func (c *IPConn) LocalAddr() Addr
```

LocalAddr returns the local network address.

func (*IPConn) [Read](#)

```
func (c *IPConn) Read(b []byte) (int, error)
```

Read implements the Conn Read method.

func (*IPConn) [ReadFrom](#)

```
func (c *IPConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom implements the PacketConn ReadFrom method.

func (*IPConn) [ReadFromIP](#)

```
func (c *IPConn) ReadFromIP(b []byte) (int, *IPAddr, error)
```

ReadFromIP reads a IP packet from c, copying the payload into b. It returns the number of bytes copied into b and the return address that was on the packet.

ReadFromIP can be made to time out and return an error with Timeout() == true after a fixed time limit; see SetDeadline and SetReadDeadline.

func (*IPConn) [RemoteAddr](#)

```
func (c *IPConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address, a *IPAddr.

func (*IPConn) [SetDeadline](#)

```
func (c *IPConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

func (*IPConn) [SetReadBuffer](#)

```
func (c *IPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

func (*IPConn) [SetReadDeadline](#)

```
func (c *IPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

func (*IPConn) [SetWriteBuffer](#)

```
func (c *IPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

func (*IPConn) [SetWriteDeadline](#)

```
func (c *IPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

func (*IPConn) [Write](#)

```
func (c *IPConn) Write(b []byte) (int, error)
```

Write implements the Conn Write method.

func (*IPConn) [WriteTo](#)

```
func (c *IPConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo implements the PacketConn WriteTo method.

func (*IPConn) [WriteToIP](#)

```
func (c *IPConn) writeToIP(b []byte, addr *IPAddr) (int, error)
```

WriteToIP writes a IP packet to addr via c, copying the payload from b.

WriteToIP can be made to time out and return an error with Timeout() == true after a fixed time limit; see SetDeadline and SetWriteDeadline. On packet-oriented connections, write timeouts are rare.

type [IPMask](#)

type IPMask []byte

An IP mask is an IP address.

func [CIDRMask](#)

func CIDRMask(ones, bits int) IPMask

CIDRMask returns an IPMask consisting of `ones` 1 bits followed by 0s up to a total length of `bits` bits. For a mask of this form, CIDRMask is the inverse of IPMask.Size.

func [IPv4Mask](#)

func IPv4Mask(a, b, c, d byte) IPMask

IPv4Mask returns the IP mask (in 4-byte form) of the IPv4 mask a.b.c.d.

func (IPMask) [Size](#)

func (m IPMask) Size() (ones, bits int)

Size returns the number of leading ones and total bits in the mask. If the mask is not in the canonical form--ones followed by zeros--then Size returns 0, 0.

func (IPMask) [String](#)

func (m IPMask) String() string

String returns the hexadecimal form of m, with no punctuation.

type [IPNet](#)

```
type IPNet struct {
    IP    IP    // network number
    Mask IPMask // network mask
}
```

An IPNet represents an IP network.

func (*IPNet) [Contains](#)

```
func (n *IPNet) Contains(ip IP) bool
```

Contains reports whether the network includes ip.

func (*IPNet) [Network](#)

```
func (n *IPNet) Network() string
```

Network returns the address's network name, "ip+net".

func (*IPNet) [String](#)

```
func (n *IPNet) String() string
```

String returns the CIDR notation of n like "192.168.100.1/24" or "2001:DB8::/48" as defined in RFC 4632 and RFC 4291. If the mask is not in the canonical form, it returns the string which consists of an IP address, followed by a slash character and a mask expressed as hexadecimal form with no punctuation like "192.168.100.1/c000ff00".

type [Interface](#)

```
type Interface struct {
    Index      int           // positive integer that starts at one
    MTU        int           // maximum transmission unit
    Name       string        // e.g., "en0", "lo0", "eth0.100"
    HardwareAddr HardwareAddr // IEEE MAC-48, EUI-48 and EUI-64 form
    Flags      Flags         // e.g., FlagUp, FlagLoopback, FlagMul
}
```

Interface represents a mapping between network interface name and index. It also represents network interface facility information.

func [InterfaceByIndex](#)

```
func InterfaceByIndex(index int) (*Interface, error)
```

InterfaceByIndex returns the interface specified by index.

func [InterfaceByName](#)

```
func InterfaceByName(name string) (*Interface, error)
```

InterfaceByName returns the interface specified by name.

func (*Interface) [Addrs](#)

```
func (ifi *Interface) Addrs() ([]Addr, error)
```

Addrs returns interface addresses for a specific interface.

func (*Interface) [MulticastAddrs](#)

```
func (ifi *Interface) MulticastAddrs() ([]Addr, error)
```

MulticastAddrs returns multicast, joined group addresses for a specific interface.

type [InvalidAddrError](#)

```
type InvalidAddrError string
```

func (InvalidAddrError) [Error](#)

```
func (e InvalidAddrError) Error() string
```

func (InvalidAddrError) [Temporary](#)

```
func (e InvalidAddrError) Temporary() bool
```

func (InvalidAddrError) [Timeout](#)

```
func (e InvalidAddrError) Timeout() bool
```

type [Listener](#)

```
type Listener interface {
    // Accept waits for and returns the next connection to the listener.
    Accept() (c Conn, err error)

    // Close closes the listener.
    // Any blocked Accept operations will be unblocked and return errors.
    Close() error

    // Addr returns the listener's network address.
    Addr() Addr
}
```

A Listener is a generic network listener for stream-oriented protocols.

Multiple goroutines may invoke methods on a Listener simultaneously.

? Example

? Example

Code:

```
// Listen on TCP port 2000 on all interfaces.
l, err := net.Listen("tcp", ":2000")
if err != nil {
    log.Fatal(err)
}
for {
    // Wait for a connection.
    conn, err := l.Accept()
    if err != nil {
        log.Fatal(err)
    }
    // Handle the connection in a new goroutine.
    // The loop then returns to accepting, so that
    // multiple connections may be served concurrently.
    go func(c net.Conn) {
        // Echo all incoming data.
        io.Copy(c, c)
        // Shut down the connection.
        c.Close()
    }(conn)
}
```

func [FileListener](#)

```
func FileListener(f *os.File) (l Listener, err error)
```

FileListener returns a copy of the network listener corresponding to the open file `f`. It is the caller's responsibility to close `l` when finished. Closing `c` does not affect `l`, and closing `l` does not affect `c`.

func [Listen](#)

```
func Listen(net, laddr string) (Listener, error)
```

Listen announces on the local network address `laddr`. The network string `net` must be a stream-oriented network: "tcp", "tcp4", "tcp6", or "unix", or "unixpacket".

type [MX](#)

```
type MX struct {  
    Host string  
    Pref uint16  
}
```

An MX represents a single DNS MX record.

type [OpError](#)

```
type OpError struct {  
    Op    string  
    Net   string  
    Addr  Addr  
    Err   error  
}
```

func (*OpError) [Error](#)

```
func (e *OpError) Error() string
```

func (*OpError) [Temporary](#)

```
func (e *OpError) Temporary() bool
```

func (*OpError) [Timeout](#)

```
func (e *OpError) Timeout() bool
```

type [PacketConn](#)

```
type PacketConn interface {
    // ReadFrom reads a packet from the connection,
    // copying the payload into b. It returns the number of
    // bytes copied into b and the return address that
    // was on the packet.
    // ReadFrom can be made to time out and return
    // an error with Timeout() == true after a fixed time limit;
    // see SetDeadline and SetReadDeadline.
    ReadFrom(b []byte) (n int, addr Addr, err error)

    // WriteTo writes a packet with payload b to addr.
    // WriteTo can be made to time out and return
    // an error with Timeout() == true after a fixed time limit;
    // see SetDeadline and SetWriteDeadline.
    // On packet-oriented connections, write timeouts are rare.
    WriteTo(b []byte, addr Addr) (n int, err error)

    // Close closes the connection.
    // Any blocked ReadFrom or WriteTo operations will be unblocked
    Close() error

    // LocalAddr returns the local network address.
    LocalAddr() Addr

    // SetDeadline sets the read and write deadlines associated
    // with the connection.
    SetDeadline(t time.Time) error

    // SetReadDeadline sets the deadline for future Read calls.
    // If the deadline is reached, Read will fail with a timeout
    // (see type Error) instead of blocking.
    // A zero value for t means Read will not time out.
    SetReadDeadline(t time.Time) error

    // SetWriteDeadline sets the deadline for future Write calls.
    // If the deadline is reached, Write will fail with a timeout
    // (see type Error) instead of blocking.
    // A zero value for t means Write will not time out.
    // Even if write times out, it may return n > 0, indicating that
    // some of the data was successfully written.
    SetWriteDeadline(t time.Time) error
}
```

PacketConn is a generic packet-oriented network connection.

Multiple goroutines may invoke methods on a `PacketConn` simultaneously.

func [FilePacketConn](#)

```
func FilePacketConn(f *os.File) (c PacketConn, err error)
```

`FilePacketConn` returns a copy of the packet network connection corresponding to the open file `f`. It is the caller's responsibility to close `f` when finished. Closing `c` does not affect `f`, and closing `f` does not affect `c`.

func [ListenPacket](#)

```
func ListenPacket(net, addr string) (PacketConn, error)
```

`ListenPacket` announces on the local network address `laddr`. The network string `net` must be a packet-oriented network: "udp", "udp4", "udp6", "ip", "ip4", "ip6" or "unixgram".

type [ParseError](#)

```
type ParseError struct {  
    Type string  
    Text string  
}
```

A ParseError represents a malformed text string and the type of string that was expected.

func (*ParseError) [Error](#)

```
func (e *ParseError) Error() string
```

type [SRV](#)

```
type SRV struct {  
    Target    string  
    Port      uint16  
    Priority  uint16  
    Weight    uint16  
}
```

An SRV represents a single DNS SRV record.

type [TCPAddr](#)

```
type TCPAddr struct {  
    IP    IP  
    Port int  
}
```

TCPAddr represents the address of a TCP end point.

func [ResolveTCPAddr](#)

```
func ResolveTCPAddr(net, addr string) (*TCPAddr, error)
```

ResolveTCPAddr parses addr as a TCP address of the form host:port and resolves domain names or port names to numeric addresses on the network net, which must be "tcp", "tcp4" or "tcp6". A literal IPv6 host address must be enclosed in square brackets, as in "[::]:80".

func (***TCPAddr**) [Network](#)

```
func (a *TCPAddr) Network() string
```

Network returns the address's network name, "tcp".

func (***TCPAddr**) [String](#)

```
func (a *TCPAddr) String() string
```

type [TCPConn](#)

```
type TCPConn struct {  
    // contains filtered or unexported fields  
}
```

TCPConn is an implementation of the Conn interface for TCP network connections.

func [DialTCP](#)

```
func DialTCP(net string, laddr, raddr *TCPAddr) (*TCPConn, error)
```

DialTCP connects to the remote address raddr on the network net, which must be "tcp", "tcp4", or "tcp6". If laddr is not nil, it is used as the local address for the connection.

func (***TCPConn**) [Close](#)

```
func (c *TCPConn) Close() error
```

Close closes the TCP connection.

func (***TCPConn**) [CloseRead](#)

```
func (c *TCPConn) CloseRead() error
```

CloseRead shuts down the reading side of the TCP connection. Most callers should just use Close.

func (***TCPConn**) [CloseWrite](#)

```
func (c *TCPConn) CloseWrite() error
```

CloseWrite shuts down the writing side of the TCP connection. Most callers should just use Close.

func (***TCPConn**) [File](#)

```
func (c *TCPConn) File() (f *os.File, err error)
```

File returns a copy of the underlying `os.File`, set to blocking mode. It is the caller's responsibility to close `f` when finished. Closing `c` does not affect `f`, and closing `f` does not affect `c`.

func (*TCPConn) [LocalAddr](#)

```
func (c *TCPConn) LocalAddr() Addr
```

LocalAddr returns the local network address, a `*TCPAddr`.

func (*TCPConn) [Read](#)

```
func (c *TCPConn) Read(b []byte) (n int, err error)
```

Read implements the `Conn Read` method.

func (*TCPConn) [ReadFrom](#)

```
func (c *TCPConn) ReadFrom(r io.Reader) (int64, error)
```

ReadFrom implements the `io.ReaderFrom ReadFrom` method.

func (*TCPConn) [RemoteAddr](#)

```
func (c *TCPConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address, a `*TCPAddr`.

func (*TCPConn) [SetDeadline](#)

```
func (c *TCPConn) SetDeadline(t time.Time) error
```

SetDeadline implements the `Conn SetDeadline` method.

func (*TCPConn) [SetKeepAlive](#)

```
func (c *TCPConn) SetKeepAlive(keepalive bool) error
```

SetKeepAlive sets whether the operating system should send keepalive messages on the connection.

func (*TCPConn) [SetLinger](#)

```
func (c *TCPConn) SetLinger(sec int) error
```

SetLinger sets the behavior of Close() on a connection which still has data waiting to be sent or to be acknowledged.

If `sec < 0` (the default), Close returns immediately and the operating system finishes sending the data in the background.

If `sec == 0`, Close returns immediately and the operating system discards any unsent or unacknowledged data.

If `sec > 0`, Close blocks for at most `sec` seconds waiting for data to be sent and acknowledged.

func (*TCPConn) [SetNoDelay](#)

```
func (c *TCPConn) SetNoDelay(noDelay bool) error
```

SetNoDelay controls whether the operating system should delay packet transmission in hopes of sending fewer packets (Nagle's algorithm). The default is true (no delay), meaning that data is sent as soon as possible after a Write.

func (*TCPConn) [SetReadBuffer](#)

```
func (c *TCPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

func (*TCPConn) [SetReadDeadline](#)

```
func (c *TCPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

func (*TCPConn) [SetWriteBuffer](#)

```
func (c *TCPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

func (*TCPConn) [SetWriteDeadline](#)

```
func (c *TCPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

func (*TCPConn) [Write](#)

```
func (c *TCPConn) Write(b []byte) (n int, err error)
```

Write implements the Conn Write method.

type [TCPListener](#)

```
type TCPListener struct {  
    // contains filtered or unexported fields  
}
```

TCPListener is a TCP network listener. Clients should typically use variables of type `Listener` instead of assuming `TCP`.

func [ListenTCP](#)

```
func ListenTCP(net string, laddr *TCPAddr) (*TCPListener, error)
```

`ListenTCP` announces on the TCP address `laddr` and returns a TCP listener. `Net` must be `"tcp"`, `"tcp4"`, or `"tcp6"`. If `laddr` has a port of 0, it means to listen on some available port. The caller can use `l.Addr()` to retrieve the chosen address.

func (***TCPListener**) [Accept](#)

```
func (l *TCPListener) Accept() (c Conn, err error)
```

`Accept` implements the `Accept` method in the `Listener` interface; it waits for the next call and returns a generic `Conn`.

func (***TCPListener**) [AcceptTCP](#)

```
func (l *TCPListener) AcceptTCP() (c *TCPConn, err error)
```

`AcceptTCP` accepts the next incoming call and returns the new connection and the remote address.

func (***TCPListener**) [Addr](#)

```
func (l *TCPListener) Addr() Addr
```

`Addr` returns the listener's network address, a `*TCPAddr`.

func (***TCPListener**) [Close](#)

```
func (l *TCPListener) Close() error
```

Close stops listening on the TCP address. Already Accepted connections are not closed.

func (*TCPListener) [File](#)

```
func (l *TCPListener) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File, set to blocking mode. It is the caller's responsibility to close f when finished. Closing l does not affect f, and closing f does not affect l.

func (*TCPListener) [SetDeadline](#)

```
func (l *TCPListener) SetDeadline(t time.Time) error
```

SetDeadline sets the deadline associated with the listener. A zero time value disables the deadline.

type [UDPAddr](#)

```
type UDPAddr struct {  
    IP    IP  
    Port int  
}
```

UDPAddr represents the address of a UDP end point.

func [ResolveUDPAddr](#)

```
func ResolveUDPAddr(net, addr string) (*UDPAddr, error)
```

ResolveUDPAddr parses addr as a UDP address of the form host:port and resolves domain names or port names to numeric addresses on the network net, which must be "udp", "udp4" or "udp6". A literal IPv6 host address must be enclosed in square brackets, as in "[::]:80".

func (***UDPAddr**) [Network](#)

```
func (a *UDPAddr) Network() string
```

Network returns the address's network name, "udp".

func (***UDPAddr**) [String](#)

```
func (a *UDPAddr) String() string
```

type [UDPConn](#)

```
type UDPConn struct {  
    // contains filtered or unexported fields  
}
```

UDPConn is the implementation of the Conn and PacketConn interfaces for UDP network connections.

func [DialUDP](#)

```
func DialUDP(net string, laddr, raddr *UDPAddr) (*UDPConn, error)
```

DialUDP connects to the remote address raddr on the network net, which must be "udp", "udp4", or "udp6". If laddr is not nil, it is used as the local address for the connection.

func [ListenMulticastUDP](#)

```
func ListenMulticastUDP(net string, ifi *Interface, gaddr *UDPAddr)
```

ListenMulticastUDP listens for incoming multicast UDP packets addressed to the group address gaddr on ifi, which specifies the interface to join. ListenMulticastUDP uses default multicast interface if ifi is nil.

func [ListenUDP](#)

```
func ListenUDP(net string, laddr *UDPAddr) (*UDPConn, error)
```

ListenUDP listens for incoming UDP packets addressed to the local address laddr. The returned connection c's ReadFrom and WriteTo methods can be used to receive and send UDP packets with per-packet addressing.

func [ListenUnixgram](#)

```
func ListenUnixgram(net string, laddr *UnixAddr) (*UDPConn, error)
```

ListenUnixgram listens for incoming Unix datagram packets addressed to the local address laddr. The returned connection c's ReadFrom and WriteTo methods

can be used to receive and send UDP packets with per-packet addressing. The network net must be "unixgram".

func (*UDPConn) [Close](#)

```
func (c *UDPConn) Close() error
```

Close closes the UDP connection.

func (*UDPConn) [File](#)

```
func (c *UDPConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File, set to blocking mode. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

func (*UDPConn) [LocalAddr](#)

```
func (c *UDPConn) LocalAddr() Addr
```

LocalAddr returns the local network address.

func (*UDPConn) [Read](#)

```
func (c *UDPConn) Read(b []byte) (int, error)
```

Read implements the Conn Read method.

func (*UDPConn) [ReadFrom](#)

```
func (c *UDPConn) ReadFrom(b []byte) (int, Addr, error)
```

ReadFrom implements the PacketConn ReadFrom method.

func (*UDPConn) [ReadFromUDP](#)

```
func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr, err e
```

ReadFromUDP reads a UDP packet from c, copying the payload into b. It returns

the number of bytes copied into b and the return address that was on the packet.

ReadFromUDP can be made to time out and return an error with Timeout() == true after a fixed time limit; see SetDeadline and SetReadDeadline.

func (*UDPConn) [RemoteAddr](#)

```
func (c *UDPConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address, a *UDPAddr.

func (*UDPConn) [SetDeadline](#)

```
func (c *UDPConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

func (*UDPConn) [SetReadBuffer](#)

```
func (c *UDPConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

func (*UDPConn) [SetReadDeadline](#)

```
func (c *UDPConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

func (*UDPConn) [SetWriteBuffer](#)

```
func (c *UDPConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

func (*UDPConn) [SetWriteDeadline](#)

```
func (c *UDPConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

func (*UDPConn) [Write](#)

```
func (c *UDPConn) Write(b []byte) (int, error)
```

Write implements the Conn Write method.

func (*UDPConn) [WriteTo](#)

```
func (c *UDPConn) WriteTo(b []byte, addr Addr) (int, error)
```

WriteTo implements the PacketConn WriteTo method.

func (*UDPConn) [WriteToUDP](#)

```
func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int, error)
```

WriteToUDP writes a UDP packet to addr via c, copying the payload from b.

WriteToUDP can be made to time out and return an error with Timeout() == true after a fixed time limit; see SetDeadline and SetWriteDeadline. On packet-oriented connections, write timeouts are rare.

type [UnixAddr](#)

```
type UnixAddr struct {  
    Name string  
    Net  string  
}
```

UnixAddr represents the address of a Unix domain socket end point.

func [ResolveUnixAddr](#)

```
func ResolveUnixAddr(net, addr string) (*UnixAddr, error)
```

ResolveUnixAddr parses addr as a Unix domain socket address. The string net gives the network name, "unix", "unixgram" or "unixpacket".

func (*UnixAddr) [Network](#)

```
func (a *UnixAddr) Network() string
```

Network returns the address's network name, "unix" or "unixgram".

func (*UnixAddr) [String](#)

```
func (a *UnixAddr) String() string
```

type [UnixConn](#)

```
type UnixConn struct {  
    // contains filtered or unexported fields  
}
```

UnixConn is an implementation of the Conn interface for connections to Unix domain sockets.

func [DialUnix](#)

```
func DialUnix(net string, laddr, raddr *UnixAddr) (*UnixConn, error)
```

DialUnix connects to the remote address raddr on the network net, which must be "unix" or "unixgram". If laddr is not nil, it is used as the local address for the connection.

func (*UnixConn) [Close](#)

```
func (c *UnixConn) Close() error
```

Close closes the Unix domain connection.

func (*UnixConn) [File](#)

```
func (c *UnixConn) File() (f *os.File, err error)
```

File returns a copy of the underlying os.File, set to blocking mode. It is the caller's responsibility to close f when finished. Closing c does not affect f, and closing f does not affect c.

func (*UnixConn) [LocalAddr](#)

```
func (c *UnixConn) LocalAddr() Addr
```

LocalAddr returns the local network address, a *UnixAddr. Unlike in other protocols, LocalAddr is usually nil for dialed connections.

func (*UnixConn) [Read](#)

```
func (c *UnixConn) Read(b []byte) (n int, err error)
```

Read implements the Conn Read method.

func (*UnixConn) [ReadFrom](#)

```
func (c *UnixConn) ReadFrom(b []byte) (n int, addr Addr, err error)
```

ReadFrom implements the PacketConn ReadFrom method.

func (*UnixConn) [ReadFromUnix](#)

```
func (c *UnixConn) ReadFromUnix(b []byte) (n int, addr *UnixAddr, er
```

ReadFromUnix reads a packet from c, copying the payload into b. It returns the number of bytes copied into b and the source address of the packet.

ReadFromUnix can be made to time out and return an error with Timeout() == true after a fixed time limit; see SetDeadline and SetReadDeadline.

func (*UnixConn) [ReadMsgUnix](#)

```
func (c *UnixConn) ReadMsgUnix(b, oob []byte) (n, oobn, flags int, a
```

ReadMsgUnix reads a packet from c, copying the payload into b and the associated out-of-band data into oob. It returns the number of bytes copied into b, the number of bytes copied into oob, the flags that were set on the packet, and the source address of the packet.

func (*UnixConn) [RemoteAddr](#)

```
func (c *UnixConn) RemoteAddr() Addr
```

RemoteAddr returns the remote network address, a *UnixAddr. Unlike in other protocols, RemoteAddr is usually nil for connections accepted by a listener.

func (*UnixConn) [SetDeadline](#)

```
func (c *UnixConn) SetDeadline(t time.Time) error
```

SetDeadline implements the Conn SetDeadline method.

func (*UnixConn) [SetReadBuffer](#)

```
func (c *UnixConn) SetReadBuffer(bytes int) error
```

SetReadBuffer sets the size of the operating system's receive buffer associated with the connection.

func (*UnixConn) [SetReadDeadline](#)

```
func (c *UnixConn) SetReadDeadline(t time.Time) error
```

SetReadDeadline implements the Conn SetReadDeadline method.

func (*UnixConn) [SetWriteBuffer](#)

```
func (c *UnixConn) SetWriteBuffer(bytes int) error
```

SetWriteBuffer sets the size of the operating system's transmit buffer associated with the connection.

func (*UnixConn) [SetWriteDeadline](#)

```
func (c *UnixConn) SetWriteDeadline(t time.Time) error
```

SetWriteDeadline implements the Conn SetWriteDeadline method.

func (*UnixConn) [Write](#)

```
func (c *UnixConn) Write(b []byte) (n int, err error)
```

Write implements the Conn Write method.

func (*UnixConn) [WriteMsgUnix](#)

```
func (c *UnixConn) WriteMsgUnix(b, oob []byte, addr *UnixAddr) (n, o
```

WriteMsgUnix writes a packet to addr via c, copying the payload from b and the associated out-of-band data from oob. It returns the number of payload and out-

of-band bytes written.

func (*UnixConn) [WriteTo](#)

```
func (c *UnixConn) WriteTo(b []byte, addr Addr) (n int, err error)
```

WriteTo implements the PacketConn WriteTo method.

func (*UnixConn) [WriteToUnix](#)

```
func (c *UnixConn) WriteToUnix(b []byte, addr *UnixAddr) (n int, err
```

WriteToUnix writes a packet to addr via c, copying the payload from b.

WriteToUnix can be made to time out and return an error with Timeout() == true after a fixed time limit; see SetDeadline and SetWriteDeadline. On packet-oriented connections, write timeouts are rare.

type [UnixListener](#)

```
type UnixListener struct {  
    // contains filtered or unexported fields  
}
```

UnixListener is a Unix domain socket listener. Clients should typically use variables of type Listener instead of assuming Unix domain sockets.

func [ListenUnix](#)

```
func ListenUnix(net string, laddr *UnixAddr) (*UnixListener, error)
```

ListenUnix announces on the Unix domain socket laddr and returns a Unix listener. Net must be "unix" (stream sockets).

func (*UnixListener) [Accept](#)

```
func (l *UnixListener) Accept() (c Conn, err error)
```

Accept implements the Accept method in the Listener interface; it waits for the next call and returns a generic Conn.

func (*UnixListener) [AcceptUnix](#)

```
func (l *UnixListener) AcceptUnix() (*UnixConn, error)
```

AcceptUnix accepts the next incoming call and returns the new connection and the remote address.

func (*UnixListener) [Addr](#)

```
func (l *UnixListener) Addr() Addr
```

Addr returns the listener's network address.

func (*UnixListener) [Close](#)

```
func (l *UnixListener) Close() error
```

Close stops listening on the Unix address. Already accepted connections are not closed.

func (*UnixListener) [File](#)

```
func (l *UnixListener) File() (f *os.File, err error)
```

File returns a copy of the underlying `os.File`, set to blocking mode. It is the caller's responsibility to close `f` when finished. Closing `l` does not affect `f`, and closing `f` does not affect `l`.

func (*UnixListener) [SetDeadline](#)

```
func (l *UnixListener) SetDeadline(t time.Time) (err error)
```

SetDeadline sets the deadline associated with the listener. A zero time value disables the deadline.

type UnknownNetworkError

type UnknownNetworkError string

func (UnknownNetworkError) Error

func (e UnknownNetworkError) Error() string

func (UnknownNetworkError) Temporary

func (e UnknownNetworkError) Temporary() bool

func (UnknownNetworkError) Timeout

func (e UnknownNetworkError) Timeout() bool

Bugs

On OpenBSD, listening on the "tcp" network does not listen for both IPv4 and IPv6 connections. This is due to the fact that IPv4 traffic will not be routed to an IPv6 socket - two separate sockets are required if both AFs are to be supported. See `inet6(4)` on OpenBSD for details.

Subdirectories

Name	Synopsis
http	Package http provides HTTP client and server implementations.
cgi	Package cgi implements CGI (Common Gateway Interface) as specified in RFC 3875.
fcgi	Package fcgi implements the FastCGI protocol.
httpptest	Package httpptest provides utilities for HTTP testing.
httputil	Package httputil provides HTTP utility functions, complementing the more common ones in the net/http package.
pprof	Package pprof serves via its HTTP server runtime profiling data in the format expected by the pprof visualization tool.
mail	Package mail implements parsing of mail messages.
rpc	Package rpc provides access to the exported methods of an object across a network or other I/O connection.
jsonrpc	Package jsonrpc implements a JSON-RPC ClientCodec and ServerCodec for the rpc package.
smtp	Package smtp implements the Simple Mail Transfer Protocol as defined in RFC 5321.
textproto	Package textproto implements generic support for text-based request/response protocols in the style of HTTP, NNTP, and SMTP.
url	Package url parses URLs and implements query escaping.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package http

```
import "net/http"
```

[Overview](#)

[Index](#)

[Examples](#)

[Subdirectories](#)

Overview ?

Overview ?

Package `http` provides HTTP client and server implementations.

`Get`, `Head`, `Post`, and `PostForm` make HTTP requests:

```
resp, err := http.Get("http://example.com/")
...
resp, err := http.Post("http://example.com/upload", "image/jpeg", &b
...
resp, err := http.PostForm("http://example.com/form",
    url.Values{"key": {"Value"}, "id": {"123"}})
```

The client must close the response body when finished with it:

```
resp, err := http.Get("http://example.com/")
if err != nil {
    // handle error
}
defer resp.Body.Close()
body, err := ioutil.ReadAll(resp.Body)
// ...
```

For control over HTTP client headers, redirect policy, and other settings, create a `Client`:

```
client := &http.Client{
    CheckRedirect: redirectPolicyFunc,
}

resp, err := client.Get("http://example.com")
// ...

req, err := http.NewRequest("GET", "http://example.com", nil)
// ...
req.Header.Add("If-None-Match", `W/"wyzzy"`)
resp, err := client.Do(req)
// ...
```

For control over proxies, TLS configuration, keep-alives, compression, and other settings, create a `Transport`:

```
tr := &http.Transport{
    TLSClientConfig: &tls.Config{RootCAs: pool},
```

```
        DisableCompression: true,
    }
    client := &http.Client{Transport: tr}
    resp, err := client.Get("https://example.com")
```

Clients and Transports are safe for concurrent use by multiple goroutines and for efficiency should only be created once and re-used.

ListenAndServe starts an HTTP server with a given address and handler. The handler is usually nil, which means to use DefaultServeMux. Handle and HandleFunc add handlers to DefaultServeMux:

```
http.Handle("/foo", fooHandler)

http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request)
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})

log.Fatal(http.ListenAndServe(":8080", nil))
```

More control over the server's behavior is available by creating a custom Server:

```
s := &http.Server{
    Addr:           ":8080",
    Handler:        myHandler,
    ReadTimeout:   10 * time.Second,
    WriteTimeout:  10 * time.Second,
    MaxHeaderBytes: 1 << 20,
}
log.Fatal(s.ListenAndServe())
```

Index

Constants

Variables

func CanonicalHeaderKey(s string) string

func DetectContentType(data []byte) string

func Error(w ResponseWriter, error string, code int)

func Handle(pattern string, handler Handler)

func HandleFunc(pattern string, handler func(ResponseWriter, *Request))

func ListenAndServe(addr string, handler Handler) error

func ListenAndServeTLS(addr string, certFile string, keyFile string, handler Handler) error

func MaxBytesReader(w ResponseWriter, r io.ReadCloser, n int64) io.ReadCloser

func NotFound(w ResponseWriter, r *Request)

func ParseHTTPVersion(vers string) (major, minor int, ok bool)

func ProxyFromEnvironment(req *Request) (*url.URL, error)

func ProxyURL(fixedURL *url.URL) func(*Request) (*url.URL, error)

func Redirect(w ResponseWriter, r *Request, urlStr string, code int)

func Serve(l net.Listener, handler Handler) error

func ServeContent(w ResponseWriter, req *Request, name string, modtime time.Time, content io.ReadSeeker)

func ServeFile(w ResponseWriter, r *Request, name string)

func SetCookie(w ResponseWriter, cookie *Cookie)

func StatusText(code int) string

type Client

func (c *Client) Do(req *Request) (resp *Response, err error)

func (c *Client) Get(url string) (r *Response, err error)

func (c *Client) Head(url string) (r *Response, err error)

func (c *Client) Post(url string, bodyType string, body io.Reader) (r *Response, err error)

func (c *Client) PostForm(url string, data url.Values) (r *Response, err error)

type Cookie

func (c *Cookie) String() string

type CookieJar

type Dir

[func \(d Dir\) Open\(name string\) \(File, error\)](#)
[type File](#)
[type FileSystem](#)
[type Flusher](#)
[type Handler](#)
[func FileServer\(root FileSystem\) Handler](#)
[func NotFoundHandler\(\) Handler](#)
[func RedirectHandler\(url string, code int\) Handler](#)
[func StripPrefix\(prefix string, h Handler\) Handler](#)
[func TimeoutHandler\(h Handler, dt time.Duration, msg string\) Handler](#)
[type HandlerFunc](#)
[func \(f HandlerFunc\) ServeHTTP\(w ResponseWriter, r *Request\)](#)
[type Header](#)
[func \(h Header\) Add\(key, value string\)](#)
[func \(h Header\) Del\(key string\)](#)
[func \(h Header\) Get\(key string\) string](#)
[func \(h Header\) Set\(key, value string\)](#)
[func \(h Header\) Write\(w io.Writer\) error](#)
[func \(h Header\) WriteSubset\(w io.Writer, exclude map\[string\]bool\) error](#)
[type Hijacker](#)
[type ProtocolError](#)
[func \(err *ProtocolError\) Error\(\) string](#)
[type Request](#)
[func NewRequest\(method, urlStr string, body io.Reader\) \(*Request, error\)](#)
[func ReadRequest\(b *bufio.Reader\) \(req *Request, err error\)](#)
[func \(r *Request\) AddCookie\(c *Cookie\)](#)
[func \(r *Request\) Cookie\(name string\) \(*Cookie, error\)](#)
[func \(r *Request\) Cookies\(\) \[\]*Cookie](#)
[func \(r *Request\) FormFile\(key string\) \(multipart.File, *multipart.FileHeader, error\)](#)
[func \(r *Request\) FormValue\(key string\) string](#)
[func \(r *Request\) MultipartReader\(\) \(*multipart.Reader, error\)](#)
[func \(r *Request\) ParseForm\(\) \(err error\)](#)
[func \(r *Request\) ParseMultipartForm\(maxMemory int64\) error](#)
[func \(r *Request\) ProtoAtLeast\(major, minor int\) bool](#)
[func \(r *Request\) Referer\(\) string](#)
[func \(r *Request\) SetBasicAuth\(username, password string\)](#)
[func \(r *Request\) UserAgent\(\) string](#)

[func \(r *Request\) Write\(w io.Writer\) error](#)
[func \(r *Request\) WriteProxy\(w io.Writer\) error](#)
[type Response](#)
[func Get\(url string\) \(r *Response, err error\)](#)
[func Head\(url string\) \(r *Response, err error\)](#)
[func Post\(url string, bodyType string, body io.Reader\) \(r *Response, err error\)](#)
[func PostForm\(url string, data url.Values\) \(r *Response, err error\)](#)
[func ReadResponse\(r *bufio.Reader, req *Request\) \(resp *Response, err error\)](#)
[func \(r *Response\) Cookies\(\) \[\]*Cookie](#)
[func \(r *Response\) Location\(\) \(*url.URL, error\)](#)
[func \(r *Response\) ProtoAtLeast\(major, minor int\) bool](#)
[func \(r *Response\) Write\(w io.Writer\) error](#)
[type ResponseWriter](#)
[type RoundTripper](#)
[func NewFileTransport\(fs FileSystem\) RoundTripper](#)
[type ServeMux](#)
[func NewServeMux\(\) *ServeMux](#)
[func \(mux *ServeMux\) Handle\(pattern string, handler Handler\)](#)
[func \(mux *ServeMux\) HandleFunc\(pattern string, handler func\(ResponseWriter, *Request\)\)](#)
[func \(mux *ServeMux\) ServeHTTP\(w ResponseWriter, r *Request\)](#)
[type Server](#)
[func \(srv *Server\) ListenAndServe\(\) error](#)
[func \(srv *Server\) ListenAndServeTLS\(certFile, keyFile string\) error](#)
[func \(srv *Server\) Serve\(l net.Listener\) error](#)
[type Transport](#)
[func \(t *Transport\) CloseIdleConnections\(\)](#)
[func \(t *Transport\) RegisterProtocol\(scheme string, rt RoundTripper\)](#)
[func \(t *Transport\) RoundTrip\(req *Request\) \(resp *Response, err error\)](#)

Examples

[FileServer](#)

[Get](#)

[Hijacker](#)

Package files

[chunked.go](#) [client.go](#) [cookie.go](#) [doc.go](#) [filetransport.go](#) [fs.go](#) [header.go](#) [jar.go](#) [lex.go](#) [request.go](#) [response.go](#)
[server.go](#) [sniff.go](#) [status.go](#) [transfer.go](#) [transport.go](#)

Constants

```
const (  
    StatusContinue          = 100  
    StatusSwitchingProtocols = 101  
  
    StatusOK                = 200  
    StatusCreated           = 201  
    StatusAccepted          = 202  
    StatusNonAuthoritativeInfo = 203  
    StatusNoContent         = 204  
    StatusResetContent      = 205  
    StatusPartialContent    = 206  
  
    StatusMultipleChoices   = 300  
    StatusMovedPermanently  = 301  
    StatusFound             = 302  
    StatusSeeOther         = 303  
    StatusNotModified       = 304  
    StatusUseProxy          = 305  
    StatusTemporaryRedirect = 307  
  
    StatusBadRequest        = 400  
    StatusUnauthorized       = 401  
    StatusPaymentRequired   = 402  
    StatusForbidden         = 403  
    StatusNotFound          = 404  
    StatusMethodNotAllowed  = 405  
    StatusNotAcceptable     = 406  
    StatusProxyAuthRequired = 407  
    StatusRequestTimeout    = 408  
    StatusConflict          = 409  
    StatusGone              = 410  
    StatusLengthRequired    = 411  
    StatusPreconditionFailed = 412  
    StatusRequestEntityTooLarge = 413  
    StatusRequestURITooLong = 414  
    StatusUnsupportedMediaType = 415  
    StatusRequestedRangeNotSatisfiable = 416  
    StatusExpectationFailed = 417  
    StatusTeapot            = 418  
  
    StatusInternalServerError = 500  
    StatusNotImplemented      = 501  
    StatusBadGateway          = 502  
    StatusServiceUnavailable  = 503  
    StatusGatewayTimeout      = 504
```

```
    StatusHTTPVersionNotSupported = 505
)
```

HTTP status codes, defined in RFC 2616.

```
const DefaultMaxHeaderBytes = 1 << 20 // 1 MB
```

DefaultMaxHeaderBytes is the maximum permitted size of the headers in an HTTP request. This can be overridden by setting `Server.MaxHeaderBytes`.

```
const DefaultMaxIdleConnsPerHost = 2
```

DefaultMaxIdleConnsPerHost is the default value of Transport's `MaxIdleConnsPerHost`.

```
const TimeFormat = "Mon, 02 Jan 2006 15:04:05 GMT"
```

TimeFormat is the time format to use with `time.Parse` and `time.Time.Format` when parsing or generating times in HTTP headers. It is like `time.RFC1123` but hard codes GMT as the time zone.

Variables

```
var (
    ErrHeaderTooLong      = &ProtocolError{"header too long"}
    ErrShortBody          = &ProtocolError{"entity body too short"}
    ErrNotSupported       = &ProtocolError{"feature not supported"}
    ErrUnexpectedTrailer  = &ProtocolError{"trailer header without"}
    ErrMissingContentLength = &ProtocolError{"missing ContentLength"}
    ErrNotMultipart       = &ProtocolError{"request Content-Type i"}
    ErrMissingBoundary    = &ProtocolError{"no multipart boundary"}
)

var (
    ErrWriteAfterFlush = errors.New("Conn.Write called after Flush")
    ErrBodyNotAllowed  = errors.New("http: response status code does")
    ErrHijacked        = errors.New("Conn has been hijacked")
    ErrContentLength   = errors.New("Conn.Write wrote more than the")
)
```

Errors introduced by the HTTP server.

```
var DefaultClient = &Client{}
```

DefaultClient is the default Client and is used by Get, Head, and Post.

```
var DefaultServeMux = NewServeMux()
```

DefaultServeMux is the default ServeMux used by Serve.

```
var ErrBodyReadAfterClose = errors.New("http: invalid Read on closed")
```

ErrBodyReadAfterClose is returned when reading a Request Body after the body has been closed. This typically happens when the body is read after an HTTP Handler calls WriteHeader or Write on its ResponseWriter.

```
var ErrHandlerTimeout = errors.New("http: Handler timeout")
```

ErrHandlerTimeout is returned on ResponseWriter Write calls in handlers which have timed out.

```
var ErrLineTooLong = errors.New("header line too long")
```

```
var ErrMissingFile = errors.New("http: no such file")
```

ErrMissingFile is returned by FormFile when the provided file field name is either not present in the request or not a file field.

```
var ErrNoCookie = errors.New("http: named cookie not present")
```

```
var ErrNoLocation = errors.New("http: no Location header in response")
```

func [CanonicalHeaderKey](#)

```
func CanonicalHeaderKey(s string) string
```

CanonicalHeaderKey returns the canonical format of the header key s. The canonicalization converts the first letter and any letter following a hyphen to upper case; the rest are converted to lowercase. For example, the canonical key for "accept-encoding" is "Accept-Encoding".

func DetectContentType

```
func DetectContentType(data []byte) string
```

DetectContentType implements the algorithm described at <http://mimesniff.spec.whatwg.org/> to determine the Content-Type of the given data. It considers at most the first 512 bytes of data. DetectContentType always returns a valid MIME type: if it cannot determine a more specific one, it returns "application/octet-stream".

func Error

```
func Error(w ResponseWriter, error string, code int)
```

Error replies to the request with the specified error message and HTTP code.

func [Handle](#)

```
func Handle(pattern string, handler Handler)
```

Handle registers the handler for the given pattern in the DefaultServeMux. The documentation for ServeMux explains how patterns are matched.

func [HandleFunc](#)

```
func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
```

HandleFunc registers the handler function for the given pattern in the DefaultServeMux. The documentation for ServeMux explains how patterns are matched.

func [ListenAndServe](#)

```
func ListenAndServe(addr string, handler Handler) error
```

`ListenAndServe` listens on the TCP network address `addr` and then calls `Serve` with `handler` to handle requests on incoming connections. `Handler` is typically `nil`, in which case the `DefaultServeMux` is used.

A trivial example server is:

```
package main

import (
    "io"
    "net/http"
    "log"
)

// hello world, the web server
func HelloServer(w http.ResponseWriter, req *http.Request) {
    io.WriteString(w, "hello, world!\n")
}

func main() {
    http.HandleFunc("/hello", HelloServer)
    err := http.ListenAndServe(":12345", nil)
    if err != nil {
        log.Fatal("ListenAndServe: ", err)
    }
}
```

func ListenAndServeTLS

```
func ListenAndServeTLS(addr string, certFile string, keyFile string,
```

ListenAndServeTLS acts identically to ListenAndServe, except that it expects HTTPS connections. Additionally, files containing a certificate and matching private key for the server must be provided. If the certificate is signed by a certificate authority, the certFile should be the concatenation of the server's certificate followed by the CA's certificate.

A trivial example server is:

```
import (
    "log"
    "net/http"
)

func handler(w http.ResponseWriter, req *http.Request) {
    w.Header().Set("Content-Type", "text/plain")
    w.Write([]byte("This is an example server.\n"))
}

func main() {
    http.HandleFunc("/", handler)
    log.Printf("About to listen on 10443. Go to https://127.0.0.1:10443.")
    err := http.ListenAndServeTLS(":10443", "cert.pem", "key.pem")
    if err != nil {
        log.Fatal(err)
    }
}
```

One can use generate_cert.go in crypto/tls to generate cert.pem and key.pem.

func MaxBytesReader

```
func MaxBytesReader(w ResponseWriter, r io.ReadCloser, n int64) io.R
```

MaxBytesReader is similar to io.LimitReader but is intended for limiting the size of incoming request bodies. In contrast to io.LimitReader, MaxBytesReader's result is a ReadCloser, returns a non-EOF error for a Read beyond the limit, and Closes the underlying reader when its Close method is called.

MaxBytesReader prevents clients from accidentally or maliciously sending a large request and wasting server resources.

func NotFound

```
func NotFound(w ResponseWriter, r *Request)
```

NotFound replies to the request with an HTTP 404 not found error.

func ParseHTTPVersion

```
func ParseHTTPVersion(vers string) (major, minor int, ok bool)
```

ParseHTTPVersion parses a HTTP version string. "HTTP/1.0" returns (1, 0, true).

func ProxyFromEnvironment

```
func ProxyFromEnvironment(req *Request) (*url.URL, error)
```

ProxyFromEnvironment returns the URL of the proxy to use for a given request, as indicated by the environment variables \$HTTP_PROXY and \$NO_PROXY (or \$http_proxy and \$no_proxy). An error is returned if the proxy environment is invalid. A nil URL and nil error are returned if no proxy is defined in the environment, or a proxy should not be used for the given request.

func [ProxyURL](#)

```
func ProxyURL(fixedURL *url.URL) func(*Request) (*url.URL, error)
```

ProxyURL returns a proxy function (for use in a Transport) that always returns the same URL.

func [Redirect](#)

```
func Redirect(w ResponseWriter, r *Request, urlStr string, code int)
```

Redirect replies to the request with a redirect to url, which may be a path relative to the request path.

func [Serve](#)

```
func Serve(l net.Listener, handler Handler) error
```

Serve accepts incoming HTTP connections on the listener `l`, creating a new service thread for each. The service threads read requests and then call `handler` to reply to them. `Handler` is typically `nil`, in which case the `DefaultServeMux` is used.

func [ServeContent](#)

```
func ServeContent(w ResponseWriter, req *Request, name string, modti
```

ServeContent replies to the request using the content in the provided `ReadSeeker`. The main benefit of `ServeContent` over `io.Copy` is that it handles Range requests properly, sets the MIME type, and handles If-Modified-Since requests.

If the response's Content-Type header is not set, `ServeContent` first tries to deduce the type from name's file extension and, if that fails, falls back to reading the first block of the content and passing it to `DetectContentType`. The name is otherwise unused; in particular it can be empty and is never sent in the response.

If `modtime` is not the zero time, `ServeContent` includes it in a Last-Modified header in the response. If the request includes an If-Modified-Since header, `ServeContent` uses `modtime` to decide whether the content needs to be sent at all.

The content's `Seek` method must work: `ServeContent` uses a seek to the end of the content to determine its size.

Note that `*os.File` implements the `io.ReadSeeker` interface.

func [ServeFile](#)

```
func ServeFile(w ResponseWriter, r *Request, name string)
```

ServeFile replies to the request with the contents of the named file or directory.

func [SetCookie](#)

```
func SetCookie(w ResponseWriter, cookie *Cookie)
```

SetCookie adds a Set-Cookie header to the provided ResponseWriter's headers.

func [StatusText](#)

```
func StatusText(code int) string
```

StatusText returns a text for the HTTP status code. It returns the empty string if the code is unknown.

type [Client](#)

```
type Client struct {
    // Transport specifies the mechanism by which individual
    // HTTP requests are made.
    // If nil, DefaultTransport is used.
    Transport RoundTripper

    // CheckRedirect specifies the policy for handling redirects.
    // If CheckRedirect is not nil, the client calls it before
    // following an HTTP redirect. The arguments req and via
    // are the upcoming request and the requests made already,
    // oldest first. If CheckRedirect returns an error, the client
    // returns that error instead of issue the Request req.
    //
    // If CheckRedirect is nil, the Client uses its default policy,
    // which is to stop after 10 consecutive requests.
    CheckRedirect func(req *Request, via []*Request) error

    // Jar specifies the cookie jar.
    // If Jar is nil, cookies are not sent in requests and ignored
    // in responses.
    Jar CookieJar
}
```

A Client is an HTTP client. Its zero value (DefaultClient) is a usable client that uses DefaultTransport.

The Client's Transport typically has internal state (cached TCP connections), so Clients should be reused instead of created as needed. Clients are safe for concurrent use by multiple goroutines.

func (*Client) [Do](#)

```
func (c *Client) Do(req *Request) (resp *Response, err error)
```

Do sends an HTTP request and returns an HTTP response, following policy (e.g. redirects, cookies, auth) as configured on the client.

A non-nil response always contains a non-nil resp.Body.

Callers should close resp.Body when done reading from it. If resp.Body is not closed, the Client's underlying RoundTripper (typically Transport) may not be

able to re-use a persistent TCP connection to the server for a subsequent "keep-alive" request.

Generally Get, Post, or PostForm will be used instead of Do.

func (*Client) [Get](#)

```
func (c *Client) Get(url string) (r *Response, err error)
```

Get issues a GET to the specified URL. If the response is one of the following redirect codes, Get follows the redirect after calling the Client's CheckRedirect function.

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
```

Caller should close r.Body when done reading from it.

func (*Client) [Head](#)

```
func (c *Client) Head(url string) (r *Response, err error)
```

Head issues a HEAD to the specified URL. If the response is one of the following redirect codes, Head follows the redirect after calling the Client's CheckRedirect function.

```
301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)
```

func (*Client) [Post](#)

```
func (c *Client) Post(url string, bodyType string, body io.Reader) (
```

Post issues a POST to the specified URL.

Caller should close r.Body when done reading from it.

func (*Client) [PostForm](#)

```
func (c *Client) PostForm(url string, data url.Values) (r *Response,
```

PostForm issues a POST to the specified URL, with data's keys and values urlencoded as the request body.

Caller should close r.Body when done reading from it.

type [Cookie](#)

```
type Cookie struct {
    Name      string
    Value     string
    Path      string
    Domain    string
    Expires   time.Time
    RawExpires string

    // MaxAge=0 means no 'Max-Age' attribute specified.
    // MaxAge<0 means delete cookie now, equivalently 'Max-Age: 0'
    // MaxAge>0 means Max-Age attribute present and given in seconds
    MaxAge    int
    Secure    bool
    HttpOnly  bool
    Raw       string
    Unparsed []string // Raw text of unparsed attribute-value pairs
}
```

A Cookie represents an HTTP cookie as sent in the Set-Cookie header of an HTTP response or the Cookie header of an HTTP request.

func (*Cookie) [String](#)

```
func (c *Cookie) String() string
```

String returns the serialization of the cookie for use in a Cookie header (if only Name and Value are set) or a Set-Cookie response header (if other fields are set).

type [CookieJar](#)

```
type CookieJar interface {  
    // SetCookies handles the receipt of the cookies in a reply for  
    // given URL. It may or may not choose to save the cookies, dep  
    // on the jar's policy and implementation.  
    SetCookies(u *url.URL, cookies []*Cookie)  
  
    // Cookies returns the cookies to send in a request for the give  
    // It is up to the implementation to honor the standard cookie u  
    // restrictions such as in RFC 6265.  
    Cookies(u *url.URL) []*Cookie  
}
```

A CookieJar manages storage and use of cookies in HTTP requests.

Implementations of CookieJar must be safe for concurrent use by multiple goroutines.

type [Dir](#)

```
type Dir string
```

A `Dir` implements `http.FileSystem` using the native file system restricted to a specific directory tree.

An empty `Dir` is treated as `"."`.

func ([Dir](#)) [Open](#)

```
func (d Dir) Open(name string) (File, error)
```

type [File](#)

```
type File interface {
    Close() error
    Stat() (os.FileInfo, error)
    Readdir(count int) ([]os.FileInfo, error)
    Read([]byte) (int, error)
    Seek(offset int64, whence int) (int64, error)
}
```

A File is returned by a FileSystem's Open method and can be served by the FileServer implementation.

type [FileSystem](#)

```
type FileSystem interface {  
    Open(name string) (File, error)  
}
```

A `FileSystem` implements access to a collection of named files. The elements in a file path are separated by slash ('/', U+002F) characters, regardless of host operating system convention.

type [Flusher](#)

```
type Flusher interface {  
    // Flush sends any buffered data to the client.  
    Flush()  
}
```

The Flusher interface is implemented by ResponseWriters that allow an HTTP handler to flush buffered data to the client.

Note that even for ResponseWriters that support Flush, if the client is connected through an HTTP proxy, the buffered data may not reach the client until the response completes.

type [Handler](#)

```
type Handler interface {  
    ServeHTTP(ResponseWriter, *Request)  
}
```

Objects implementing the Handler interface can be registered to serve a particular path or subtree in the HTTP server.

ServeHTTP should write reply headers and data to the ResponseWriter and then return. Returning signals that the request is finished and that the HTTP server can move on to the next request on the connection.

func [FileServer](#)

```
func FileServer(root FileSystem) Handler
```

FileServer returns a handler that serves HTTP requests with the contents of the file system rooted at root.

To use the operating system's file system implementation, use http.Dir:

```
http.Handle("/", http.FileServer(http.Dir("/tmp")))
```

? Example

? Example

Code:

```
// we use StripPrefix so that /tmpfiles/somefile will access /tmp/sc  
http.Handle("/tmpfiles/", http.StripPrefix("/tmpfiles/", http.FileSe
```

func [NotFoundHandler](#)

```
func NotFoundHandler() Handler
```

NotFoundHandler returns a simple request handler that replies to each request with a “404 page not found” reply.

func [RedirectHandler](#)

```
func RedirectHandler(url string, code int) Handler
```

RedirectHandler returns a request handler that redirects each request it receives to the given url using the given status code.

func [StripPrefix](#)

```
func StripPrefix(prefix string, h Handler) Handler
```

StripPrefix returns a handler that serves HTTP requests by removing the given prefix from the request URL's Path and invoking the handler h. StripPrefix handles a request for a path that doesn't begin with prefix by replying with an HTTP 404 not found error.

func [TimeoutHandler](#)

```
func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler
```

TimeoutHandler returns a Handler that runs h with the given time limit.

The new Handler calls h.ServeHTTP to handle each request, but if a call runs for more than ns nanoseconds, the handler responds with a 503 Service Unavailable error and the given message in its body. (If msg is empty, a suitable default message will be sent.) After such a timeout, writes by h to its ResponseWriter will return ErrHandlerTimeout.

type [HandlerFunc](#)

```
type HandlerFunc func(ResponseWriter, *Request)
```

The HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers. If `f` is a function with the appropriate signature, `HandlerFunc(f)` is a Handler object that calls `f`.

func (HandlerFunc) [ServeHTTP](#)

```
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request)
```

ServeHTTP calls `f(w, r)`.

type [Header](#)

```
type Header map[string][]string
```

A Header represents the key-value pairs in an HTTP header.

func (Header) [Add](#)

```
func (h Header) Add(key, value string)
```

Add adds the key, value pair to the header. It appends to any existing values associated with key.

func (Header) [Del](#)

```
func (h Header) Del(key string)
```

Del deletes the values associated with key.

func (Header) [Get](#)

```
func (h Header) Get(key string) string
```

Get gets the first value associated with the given key. If there are no values associated with the key, Get returns "". To access multiple values of a key, access the map directly with CanonicalHeaderKey.

func (Header) [Set](#)

```
func (h Header) Set(key, value string)
```

Set sets the header entries associated with key to the single element value. It replaces any existing values associated with key.

func (Header) [Write](#)

```
func (h Header) Write(w io.Writer) error
```

Write writes a header in wire format.

func (Header) [WriteSubset](#)

```
func (h Header) WriteSubset(w io.Writer, exclude map[string]bool) error
```

WriteSubset writes a header in wire format. If exclude is not nil, keys where exclude[key] == true are not written.

type Hijacker

```
type Hijacker interface {
    // Hijack lets the caller take over the connection.
    // After a call to Hijack(), the HTTP server library
    // will not do anything else with the connection.
    // It becomes the caller's responsibility to manage
    // and close the connection.
    Hijack() (net.Conn, *bufio.ReadWriter, error)
}
```

The Hijacker interface is implemented by ResponseWriters that allow an HTTP handler to take over the connection.

? Example

? Example

Code:

```
http.HandleFunc("/hijack", func(w http.ResponseWriter, r *http.Reque
    hj, ok := w.(http.Hijacker)
    if !ok {
        http.Error(w, "webserver doesn't support hijacking", http.St
        return
    }
    conn, bufrw, err := hj.Hijack()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    // Don't forget to close the connection:
    defer conn.Close()
    bufrw.WriteString("Now we're speaking raw TCP. Say hi: ")
    bufrw.Flush()
    s, err := bufrw.ReadString('\n')
    if err != nil {
        log.Printf("error reading string: %v", err)
        return
    }
    fmt.Fprintf(bufrw, "You said: %q\nBye.\n", s)
    bufrw.Flush()
})
```

type [ProtocolError](#)

```
type ProtocolError struct {  
    ErrorString string  
}
```

HTTP request parsing errors.

func (*ProtocolError) [Error](#)

```
func (err *ProtocolError) Error() string
```

type [Request](#)

```
type Request struct {
    Method string // GET, POST, PUT, etc.
    URL     *url.URL

    // The protocol version for incoming requests.
    // Outgoing requests always use HTTP/1.1.
    Proto      string // "HTTP/1.0"
    ProtoMajor int    // 1
    ProtoMinor int

    // A header maps request lines to their values.
    // If the header says
    //
    // accept-encoding: gzip, deflate
    // Accept-Language: en-us
    // Connection: keep-alive
    //
    // then
    //
    // Header = map[string][]string{
    //     "Accept-Encoding": {"gzip, deflate"},
    //     "Accept-Language": {"en-us"},
    //     "Connection": {"keep-alive"},
    // }
    //
    // HTTP defines that header names are case-insensitive.
    // The request parser implements this by canonicalizing the
    // name, making the first character and any characters
    // following a hyphen uppercase and the rest lowercase.
    Header Header

    // The message body.
    Body io.ReadCloser

    // ContentLength records the length of the associated content.
    // The value -1 indicates that the length is unknown.
    // Values >= 0 indicate that the given number of bytes may
    // be read from Body.
    // For outgoing requests, a value of 0 means unknown if Body is
    ContentLength int64

    // TransferEncoding lists the transfer encodings from outermost
    // innermost. An empty list denotes the "identity" encoding.
    // TransferEncoding can usually be ignored; chunked encoding is
    // automatically added and removed as necessary when sending and
```

```
// receiving requests.
TransferEncoding []string

// Close indicates whether to close the connection after
// replying to this request.
Close bool

// The host on which the URL is sought.
// Per RFC 2616, this is either the value of the Host: header
// or the host name given in the URL itself.
Host string

// Form contains the parsed form data, including both the URL
// field's query parameters and the POST or PUT form data.
// This field is only available after ParseForm is called.
// The HTTP client ignores Form and uses Body instead.
Form url.Values

// MultipartForm is the parsed multipart form, including file up
// This field is only available after ParseMultipartForm is call
// The HTTP client ignores MultipartForm and uses Body instead.
MultipartForm *multipart.Form

// Trailer maps trailer keys to values. Like for Header, if the
// response has multiple trailer lines with the same key, they w
// concatenated, delimited by commas.
// For server requests, Trailer is only populated after Body has
// closed or fully consumed.
// Trailer support is only partially complete.
Trailer Header

// RemoteAddr allows HTTP servers and other software to record
// the network address that sent the request, usually for
// logging. This field is not filled in by ReadRequest and
// has no defined format. The HTTP server in this package
// sets RemoteAddr to an "IP:port" address before invoking a
// handler.
// This field is ignored by the HTTP client.
RemoteAddr string

// RequestURI is the unmodified Request-URI of the
// Request-Line (RFC 2616, Section 5.1) as sent by the client
// to a server. Usually the URL field should be used instead.
// It is an error to set this field in an HTTP client request.
RequestURI string

// TLS allows HTTP servers and other software to record
// information about the TLS connection on which the request
// was received. This field is not filled in by ReadRequest.
```

```
    // The HTTP server in this package sets the field for
    // TLS-enabled connections before invoking a handler;
    // otherwise it leaves the field nil.
    // This field is ignored by the HTTP client.
    TLS *tls.ConnectionState
}
```

A Request represents an HTTP request received by a server or to be sent by a client.

func [NewRequest](#)

```
func NewRequest(method, urlStr string, body io.Reader) (*Request, error)
```

NewRequest returns a new Request given a method, URL, and optional body.

func [ReadRequest](#)

```
func ReadRequest(b *bufio.Reader) (req *Request, err error)
```

ReadRequest reads and parses a request from b.

func (*Request) [AddCookie](#)

```
func (r *Request) AddCookie(c *Cookie)
```

AddCookie adds a cookie to the request. Per RFC 6265 section 5.4, AddCookie does not attach more than one Cookie header field. That means all cookies, if any, are written into the same line, separated by semicolon.

func (*Request) [Cookie](#)

```
func (r *Request) Cookie(name string) (*Cookie, error)
```

Cookie returns the named cookie provided in the request or ErrNoCookie if not found.

func (*Request) [Cookies](#)

```
func (r *Request) Cookies() []*Cookie
```

Cookies parses and returns the HTTP cookies sent with the request.

func (*Request) [FormFile](#)

```
func (r *Request) FormFile(key string) (multipart.File, *multipart.F
```

FormFile returns the first file for the provided form key. FormFile calls ParseMultipartForm and ParseForm if necessary.

func (*Request) [FormValue](#)

```
func (r *Request) FormValue(key string) string
```

FormValue returns the first value for the named component of the query. FormValue calls ParseMultipartForm and ParseForm if necessary.

func (*Request) [MultipartReader](#)

```
func (r *Request) MultipartReader() (*multipart.Reader, error)
```

MultipartReader returns a MIME multipart reader if this is a multipart/form-data POST request, else returns nil and an error. Use this function instead of ParseMultipartForm to process the request body as a stream.

func (*Request) [ParseForm](#)

```
func (r *Request) ParseForm() (err error)
```

ParseForm parses the raw query from the URL.

For POST or PUT requests, it also parses the request body as a form. If the request Body's size has not already been limited by MaxBytesReader, the size is capped at 10MB.

ParseMultipartForm calls ParseForm automatically. It is idempotent.

func (*Request) [ParseMultipartForm](#)

```
func (r *Request) ParseMultipartForm(maxMemory int64) error
```

ParseMultipartForm parses a request body as multipart/form-data. The whole request body is parsed and up to a total of maxMemory bytes of its file parts are stored in memory, with the remainder stored on disk in temporary files.

ParseMultipartForm calls ParseForm if necessary. After one call to ParseMultipartForm, subsequent calls have no effect.

func (*Request) [ProtoAtLeast](#)

```
func (r *Request) ProtoAtLeast(major, minor int) bool
```

ProtoAtLeast returns whether the HTTP protocol used in the request is at least major.minor.

func (*Request) [Referer](#)

```
func (r *Request) Referer() string
```

Referer returns the referring URL, if sent in the request.

Referer is misspelled as in the request itself, a mistake from the earliest days of HTTP. This value can also be fetched from the Header map as Header["Referer"]; the benefit of making it available as a method is that the compiler can diagnose programs that use the alternate (correct English) spelling req.Referer() but cannot diagnose programs that use Header["Referrer"].

func (*Request) [SetBasicAuth](#)

```
func (r *Request) SetBasicAuth(username, password string)
```

SetBasicAuth sets the request's Authorization header to use HTTP Basic Authentication with the provided username and password.

With HTTP Basic Authentication the provided username and password are not encrypted.

func (*Request) [UserAgent](#)

```
func (r *Request) UserAgent() string
```

UserAgent returns the client's User-Agent, if sent in the request.

func (*Request) [Write](#)

```
func (r *Request) Write(w io.Writer) error
```

Write writes an HTTP/1.1 request -- header and body -- in wire format. This method consults the following fields of the request:

```
Host  
URL  
Method (defaults to "GET")  
Header  
ContentLength  
TransferEncoding  
Body
```

If Body is present, Content-Length is ≤ 0 and TransferEncoding hasn't been set to "identity", Write adds "Transfer-Encoding: chunked" to the header. Body is closed after it is sent.

func (*Request) [WriteProxy](#)

```
func (r *Request) WriteProxy(w io.Writer) error
```

WriteProxy is like Write but writes the request in the form expected by an HTTP proxy. In particular, WriteProxy writes the initial Request-URI line of the request with an absolute URI, per section 5.1.2 of RFC 2616, including the scheme and host. In either case, WriteProxy also writes a Host header, using either r.Host or r.URL.Host.

type [Response](#)

```
type Response struct {
    Status      string // e.g. "200 OK"
    StatusCode  int    // e.g. 200
    Proto       string // e.g. "HTTP/1.0"
    ProtoMajor  int    // e.g. 1
    ProtoMinor  int

    // Header maps header keys to values. If the response had multi
    // headers with the same key, they will be concatenated, with co
    // delimiters. (Section 4.2 of RFC 2616 requires that multiple
    // be semantically equivalent to a comma-delimited sequence.) Va
    // duplicated by other fields in this struct (e.g., ContentLengt
    // omitted from Header.
    //
    // Keys in the map are canonicalized (see CanonicalHeaderKey).
    Header Header

    // Body represents the response body.
    //
    // The http Client and Transport guarantee that Body is always
    // non-nil, even on responses without a body or responses with
    // a zero-lengthed body.
    Body io.ReadCloser

    // ContentLength records the length of the associated content.
    // value -1 indicates that the length is unknown. Unless Reques
    // is "HEAD", values >= 0 indicate that the given number of byte
    // be read from Body.
    ContentLength int64

    // Contains transfer encodings from outer-most to inner-most. Va
    // nil, means that "identity" encoding is used.
    TransferEncoding []string

    // Close records whether the header directed that the connection
    // closed after reading Body. The value is advice for clients:
    // ReadResponse nor Response.Write ever closes a connection.
    Close bool

    // Trailer maps trailer keys to values, in the same
    // format as the header.
    Trailer Header

    // The Request that was sent to obtain this Response.
    // Request's Body is nil (having already been consumed).
```

```
    // This is only populated for Client requests.  
    Request *Request  
}
```

Response represents the response from an HTTP request.

func [Get](#)

```
func Get(url string) (r *Response, err error)
```

Get issues a GET to the specified URL. If the response is one of the following redirect codes, Get follows the redirect, up to a maximum of 10 redirects:

```
301 (Moved Permanently)  
302 (Found)  
303 (See Other)  
307 (Temporary Redirect)
```

Caller should close r.Body when done reading from it.

Get is a wrapper around DefaultClient.Get.

? Example

? Example

Code:

```
res, err := http.Get("http://www.google.com/robots.txt")  
if err != nil {  
    log.Fatal(err)  
}  
robots, err := ioutil.ReadAll(res.Body)  
if err != nil {  
    log.Fatal(err)  
}  
res.Body.Close()  
fmt.Printf("%s", robots)
```

func [Head](#)

```
func Head(url string) (r *Response, err error)
```

Head issues a HEAD to the specified URL. If the response is one of the

following redirect codes, Head follows the redirect after calling the Client's CheckRedirect function.

301 (Moved Permanently)
302 (Found)
303 (See Other)
307 (Temporary Redirect)

Head is a wrapper around DefaultClient.Head

func [Post](#)

```
func Post(url string, bodyType string, body io.Reader) (r *Response,
```

Post issues a POST to the specified URL.

Caller should close r.Body when done reading from it.

Post is a wrapper around DefaultClient.Post

func [PostForm](#)

```
func PostForm(url string, data url.Values) (r *Response, err error)
```

PostForm issues a POST to the specified URL, with data's keys and values urlencoded as the request body.

Caller should close r.Body when done reading from it.

PostForm is a wrapper around DefaultClient.PostForm

func [ReadResponse](#)

```
func ReadResponse(r *bufio.Reader, req *Request) (resp *Response, er
```

ReadResponse reads and returns an HTTP response from r. The req parameter specifies the Request that corresponds to this Response. Clients must call resp.Body.Close when finished reading resp.Body. After that call, clients can inspect resp.Trailer to find key/value pairs included in the response trailer.

func (*Response) [Cookies](#)

```
func (r *Response) Cookies() []*Cookie
```

Cookies parses and returns the cookies set in the Set-Cookie headers.

func (*Response) [Location](#)

```
func (r *Response) Location() (*url.URL, error)
```

Location returns the URL of the response's "Location" header, if present. Relative redirects are resolved relative to the Response's Request. ErrNoLocation is returned if no Location header is present.

func (*Response) [ProtoAtLeast](#)

```
func (r *Response) ProtoAtLeast(major, minor int) bool
```

ProtoAtLeast returns whether the HTTP protocol used in the response is at least major.minor.

func (*Response) [Write](#)

```
func (r *Response) Write(w io.Writer) error
```

Writes the response (header, body and trailer) in wire format. This method consults the following fields of the response:

StatusCode

ProtoMajor

ProtoMinor

RequestMethod

TransferEncoding

Trailer

Body

ContentLength

Header, values for non-canonical keys will have unpredictable behavi

type ResponseWriter

```
type ResponseWriter interface {
    // Header returns the header map that will be sent by WriteHeader
    // Changing the header after a call to WriteHeader (or Write) has
    // no effect.
    Header() Header

    // Write writes the data to the connection as part of an HTTP response
    // If WriteHeader has not yet been called, Write calls WriteHeader
    // before writing the data. If the Header does not contain a
    // Content-Type line, Write adds a Content-Type set to the response
    // the initial 512 bytes of written data to DetectContentType.
    Write([]byte) (int, error)

    // WriteHeader sends an HTTP response header with status code.
    // If WriteHeader is not called explicitly, the first call to Write
    // will trigger an implicit WriteHeader(http.StatusOK).
    // Thus explicit calls to WriteHeader are mainly used to
    // send error codes.
    WriteHeader(int)
}
```

A ResponseWriter interface is used by an HTTP handler to construct an HTTP response.

type [RoundTripper](#)

```
type RoundTripper interface {  
    // RoundTrip executes a single HTTP transaction, returning  
    // the Response for the request req. RoundTrip should not  
    // attempt to interpret the response. In particular,  
    // RoundTrip must return err == nil if it obtained a response,  
    // regardless of the response's HTTP status code. A non-nil  
    // err should be reserved for failure to obtain a response.  
    // Similarly, RoundTrip should not attempt to handle  
    // higher-level protocol details such as redirects,  
    // authentication, or cookies.  
    //  
    // RoundTrip should not modify the request, except for  
    // consuming the Body. The request's URL and Header fields  
    // are guaranteed to be initialized.  
    RoundTrip(*Request) (*Response, error)  
}
```

RoundTripper is an interface representing the ability to execute a single HTTP transaction, obtaining the Response for a given Request.

A RoundTripper must be safe for concurrent use by multiple goroutines.

```
var DefaultTransport RoundTripper = &Transport{Proxy: ProxyFromEnvir
```

DefaultTransport is the default implementation of Transport and is used by DefaultClient. It establishes a new network connection for each call to Do and uses HTTP proxies as directed by the \$HTTP_PROXY and \$NO_PROXY (or \$http_proxy and \$no_proxy) environment variables.

func [NewFileTransport](#)

```
func NewFileTransport(fs FileSystem) RoundTripper
```

NewFileTransport returns a new RoundTripper, serving the provided FileSystem. The returned RoundTripper ignores the URL host in its incoming requests, as well as most other properties of the request.

The typical use case for NewFileTransport is to register the "file" protocol with a Transport, as in:

```
t := &http.Transport{}
t.RegisterProtocol("file", http.NewFileTransport(http.Dir("/")))
c := &http.Client{Transport: t}
res, err := c.Get("file:///etc/passwd")
...
```

type [ServeMux](#)

```
type ServeMux struct {  
    // contains filtered or unexported fields  
}
```

ServeMux is an HTTP request multiplexer. It matches the URL of each incoming request against a list of registered patterns and calls the handler for the pattern that most closely matches the URL.

Patterns named fixed, rooted paths, like `"/favicon.ico"`, or rooted subtrees, like `"/images/"` (note the trailing slash). Longer patterns take precedence over shorter ones, so that if there are handlers registered for both `"/images/"` and `"/images/thumbnails/"`, the latter handler will be called for paths beginning `"/images/thumbnails/"` and the former will receive requests for any other paths in the `"/images/"` subtree.

Patterns may optionally begin with a host name, restricting matches to URLs on that host only. Host-specific patterns take precedence over general patterns, so that a handler might register for the two patterns `"/codesearch"` and `"codesearch.google.com/"` without also taking over requests for ["http://www.google.com/"](http://www.google.com/).

ServeMux also takes care of sanitizing the URL request path, redirecting any request containing `.` or `..` elements to an equivalent `.-` and `..-` free URL.

func [NewServeMux](#)

```
func NewServeMux() *ServeMux
```

NewServeMux allocates and returns a new ServeMux.

func (*ServeMux) [Handle](#)

```
func (mux *ServeMux) Handle(pattern string, handler Handler)
```

Handle registers the handler for the given pattern. If a handler already exists for pattern, Handle panics.

func (*ServeMux) [HandleFunc](#)

```
func (mux *ServeMux) HandleFunc(pattern string, handler func(Response
```

HandleFunc registers the handler function for the given pattern.

func (*ServeMux) [ServeHTTP](#)

```
func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
```

ServeHTTP dispatches the request to the handler whose pattern most closely matches the request URL.

type [Server](#)

```
type Server struct {
    Addr          string          // TCP address to listen on, ":http"
    Handler       Handler        // handler to invoke, http.DefaultS
    ReadTimeout   time.Duration  // maximum duration before timing o
    WriteTimeout  time.Duration  // maximum duration before timing o
    MaxHeaderBytes int            // maximum size of request headers,
    TLSConfig     *tls.Config    // optional TLS config, used by Lis
}
```

A `Server` defines parameters for running an HTTP server.

func ([*Server](#)) [ListenAndServe](#)

```
func (srv *Server) ListenAndServe() error
```

`ListenAndServe` listens on the TCP network address `srv.Addr` and then calls `Serve` to handle requests on incoming connections. If `srv.Addr` is blank, `":http"` is used.

func ([*Server](#)) [ListenAndServeTLS](#)

```
func (srv *Server) ListenAndServeTLS(certFile, keyFile string) error
```

`ListenAndServeTLS` listens on the TCP network address `srv.Addr` and then calls `Serve` to handle requests on incoming TLS connections.

Filename containing a certificate and matching private key for the server must be provided. If the certificate is signed by a certificate authority, the `certFile` should be the concatenation of the server's certificate followed by the CA's certificate.

If `srv.Addr` is blank, `":https"` is used.

func ([*Server](#)) [Serve](#)

```
func (srv *Server) Serve(l net.Listener) error
```

`Serve` accepts incoming connections on the `Listener l`, creating a new service

thread for each. The service threads read requests and then call `srv.Handler` to reply to them.

type [Transport](#)

```
type Transport struct {  
  
    // Proxy specifies a function to return a proxy for a given  
    // Request. If the function returns a non-nil error, the  
    // request is aborted with the provided error.  
    // If Proxy is nil or returns a nil *URL, no proxy is used.  
    Proxy func(*Request) (*url.URL, error)  
  
    // Dial specifies the dial function for creating TCP  
    // connections.  
    // If Dial is nil, net.Dial is used.  
    Dial func(net, addr string) (c net.Conn, err error)  
  
    // TLSClientConfig specifies the TLS configuration to use with  
    // tls.Client. If nil, the default configuration is used.  
    TLSClientConfig *tls.Config  
  
    DisableKeepAlives bool  
    DisableCompression bool  
  
    // MaxIdleConnsPerHost, if non-zero, controls the maximum idle  
    // (keep-alive) to keep to keep per-host. If zero,  
    // DefaultMaxIdleConnsPerHost is used.  
    MaxIdleConnsPerHost int  
    // contains filtered or unexported fields  
}
```

Transport is an implementation of RoundTripper that supports http, https, and http proxies (for either http or https with CONNECT). Transport can also cache connections for future re-use.

func (*Transport) [CloseIdleConnections](#)

```
func (t *Transport) CloseIdleConnections()
```

CloseIdleConnections closes any connections which were previously connected from previous requests but are now sitting idle in a "keep-alive" state. It does not interrupt any connections currently in use.

func (*Transport) [RegisterProtocol](#)

```
func (t *Transport) RegisterProtocol(scheme string, rt RoundTripper)
```

RegisterProtocol registers a new protocol with scheme. The Transport will pass requests using the given scheme to rt. It is rt's responsibility to simulate HTTP request semantics.

RegisterProtocol can be used by other packages to provide implementations of protocol schemes like "ftp" or "file".

func (*Transport) [RoundTrip](#)

```
func (t *Transport) RoundTrip(req *Request) (resp *Response, err error)
```

RoundTrip implements the RoundTripper interface.

Subdirectories

Name	Synopsis
cgi	Package cgi implements CGI (Common Gateway Interface) as specified in RFC 3875.
fcgi	Package fcgi implements the FastCGI protocol.
httpptest	Package httpptest provides utilities for HTTP testing.
httputil	Package httputil provides HTTP utility functions, complementing the more common ones in the net/http package.
pprof	Package pprof serves via its HTTP server runtime profiling data in the format expected by the pprof visualization tool.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package cgi

```
import "net/http/cgi"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package cgi implements CGI (Common Gateway Interface) as specified in RFC 3875.

Note that using CGI means starting a new process to handle each request, which is typically less efficient than using a long-running server. This package is intended primarily for compatibility with existing systems.

Index

[func Request\(\) \(*http.Request, error\)](#)

[func RequestFromMap\(params map\[string\]string\) \(*http.Request, error\)](#)

[func Serve\(handler http.Handler\) error](#)

[type Handler](#)

[func \(h *Handler\) ServeHTTP\(rw http.ResponseWriter, req *http.Request\)](#)

Package files

[child.go](#) [host.go](#)

func [Request](#)

```
func Request() (*http.Request, error)
```

Request returns the HTTP request as represented in the current environment. This assumes the current program is being run by a web server in a CGI environment. The returned Request's Body is populated, if applicable.

func [RequestFromMap](#)

```
func RequestFromMap(params map[string]string) (*http.Request, error)
```

RequestFromMap creates an http.Request from CGI variables. The returned Request's Body field is not populated.

func [Serve](#)

```
func Serve(handler http.Handler) error
```

Serve executes the provided Handler on the currently active CGI request, if any. If there's no current CGI environment an error is returned. The provided handler may be nil to use `http.DefaultServeMux`.

type [Handler](#)

```
type Handler struct {
    Path string // path to the CGI executable
    Root string

    // Dir specifies the CGI executable's working directory.
    // If Dir is empty, the base directory of Path is used.
    // If Path has no base directory, the current working
    // directory is used.
    Dir string

    Env      []string // extra environment variables to set, if
    InheritEnv []string // environment variables to inherit from
    Logger    *log.Logger // optional log for errors or nil to use
    Args     []string

    // PathLocationHandler specifies the root http Handler that
    // should handle internal redirects when the CGI process
    // returns a Location header value starting with a "/", as
    // specified in RFC 3875 6.3.2. This will likely be
    // http.DefaultServeMux.
    //
    // If nil, a CGI response with a local URI path is instead sent
    // back to the client and not redirected internally.
    PathLocationHandler http.Handler
}
```

Handler runs an executable in a subprocess with a CGI environment.

func (*Handler) [ServeHTTP](#)

```
func (h *Handler) ServeHTTP(rw http.ResponseWriter, req *http.Request)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package fcgi

```
import "net/http/fcgi"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package fcgi implements the FastCGI protocol. Currently only the responder role is supported. The protocol is defined at

<http://www.fastcgi.com/drupal/node/6?q=node/22>

Index

[func Serve\(l net.Listener, handler http.Handler\) error](#)

Package files

[child.go](#) [fcgi.go](#)

func [Serve](#)

```
func Serve(l net.Listener, handler http.Handler) error
```

Serve accepts incoming FastCGI connections on the listener `l`, creating a new goroutine for each. The goroutine reads requests and then calls `handler` to reply to them. If `l` is `nil`, Serve accepts connections from `os.Stdin`. If `handler` is `nil`, `http.DefaultServeMux` is used.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package httptest

```
import "net/http/httptest"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package httptest provides utilities for HTTP testing.

Index

Constants

type ResponseRecorder

func NewRecorder() *ResponseRecorder

func (rw *ResponseRecorder) Flush()

func (rw *ResponseRecorder) Header() http.Header

func (rw *ResponseRecorder) Write(buf []byte) (int, error)

func (rw *ResponseRecorder) WriteHeader(code int)

type Server

func NewServer(handler http.Handler) *Server

func NewTLSHandler(handler http.Handler) *Server

func NewUnstartedServer(handler http.Handler) *Server

func (s *Server) Close()

func (s *Server) CloseClientConnections()

func (s *Server) Start()

func (s *Server) StartTLS()

Package files

[recorder.go](#) [server.go](#)

Constants

```
const DefaultRemoteAddr = "1.2.3.4"
```

DefaultRemoteAddr is the default remote address to return in RemoteAddr if an explicit DefaultRemoteAddr isn't set on ResponseRecorder.

type [ResponseRecorder](#)

```
type ResponseRecorder struct {  
    Code      int           // the HTTP response code from WriteHead  
    HeaderMap http.Header  // the HTTP response headers  
    Body      *bytes.Buffer // if non-nil, the bytes.Buffer to appen  
    Flushed   bool  
}
```

ResponseRecorder is an implementation of `http.ResponseWriter` that records its mutations for later inspection in tests.

func [NewRecorder](#)

```
func NewRecorder() *ResponseRecorder
```

NewRecorder returns an initialized ResponseRecorder.

func (*ResponseRecorder) [Flush](#)

```
func (rw *ResponseRecorder) Flush()
```

Flush sets `rw.Flushed` to true.

func (*ResponseRecorder) [Header](#)

```
func (rw *ResponseRecorder) Header() http.Header
```

Header returns the response headers.

func (*ResponseRecorder) [Write](#)

```
func (rw *ResponseRecorder) Write(buf []byte) (int, error)
```

Write always succeeds and writes to `rw.Body`, if not nil.

func (*ResponseRecorder) [WriteHeader](#)

```
func (rw *ResponseRecorder) WriteHeader(code int)
```

WriteHeader sets rw.Code.

type [Server](#)

```
type Server struct {
    URL      string // base URL of form http://ipaddr:port with no t
    Listener net.Listener
    TLS      *tls.Config

    // Config may be changed after calling NewUnstartedServer and
    // before Start or StartTLS.
    Config *http.Server
    // contains filtered or unexported fields
}
```

A `Server` is an HTTP server listening on a system-chosen port on the local loopback interface, for use in end-to-end HTTP tests.

func [NewServer](#)

```
func NewServer(handler http.Handler) *Server
```

`NewServer` starts and returns a new `Server`. The caller should call `Close` when finished, to shut it down.

func [NewTLSServer](#)

```
func NewTLSServer(handler http.Handler) *Server
```

`NewTLSServer` starts and returns a new `Server` using TLS. The caller should call `Close` when finished, to shut it down.

func [NewUnstartedServer](#)

```
func NewUnstartedServer(handler http.Handler) *Server
```

`NewUnstartedServer` returns a new `Server` but doesn't start it.

After changing its configuration, the caller should call `Start` or `StartTLS`.

The caller should call `Close` when finished, to shut it down.

func (*Server) [Close](#)

```
func (s *Server) Close()
```

Close shuts down the server and blocks until all outstanding requests on this server have completed.

func (*Server) [CloseClientConnections](#)

```
func (s *Server) CloseClientConnections()
```

CloseClientConnections closes any currently open HTTP connections to the test Server.

func (*Server) [Start](#)

```
func (s *Server) Start()
```

Start starts a server from NewUnstartedServer.

func (*Server) [StartTLS](#)

```
func (s *Server) StartTLS()
```

StartTLS starts TLS on a server from NewUnstartedServer.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package httputil

```
import "net/http/httputil"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `httputil` provides HTTP utility functions, complementing the more common ones in the `net/http` package.

Index

Variables

[func DumpRequest\(req *http.Request, body bool\) \(dump \[\]byte, err error\)](#)

[func DumpRequestOut\(req *http.Request, body bool\) \(\[\]byte, error\)](#)

[func DumpResponse\(resp *http.Response, body bool\) \(dump \[\]byte, err error\)](#)

[func NewChunkedReader\(r io.Reader\) io.Reader](#)

[func NewChunkedWriter\(w io.Writer\) io.WriteCloser](#)

[type ClientConn](#)

[func NewClientConn\(c net.Conn, r *bufio.Reader\) *ClientConn](#)

[func NewProxyClientConn\(c net.Conn, r *bufio.Reader\) *ClientConn](#)

[func \(cc *ClientConn\) Close\(\) error](#)

[func \(cc *ClientConn\) Do\(req *http.Request\) \(resp *http.Response, err error\)](#)

[func \(cc *ClientConn\) Hijack\(\) \(c net.Conn, r *bufio.Reader\)](#)

[func \(cc *ClientConn\) Pending\(\) int](#)

[func \(cc *ClientConn\) Read\(req *http.Request\) \(resp *http.Response, err error\)](#)

[func \(cc *ClientConn\) Write\(req *http.Request\) \(err error\)](#)

[type ReverseProxy](#)

[func NewSingleHostReverseProxy\(target *url.URL\) *ReverseProxy](#)

[func \(p *ReverseProxy\) ServeHTTP\(rw http.ResponseWriter, req *http.Request\)](#)

[type ServerConn](#)

[func NewServerConn\(c net.Conn, r *bufio.Reader\) *ServerConn](#)

[func \(sc *ServerConn\) Close\(\) error](#)

[func \(sc *ServerConn\) Hijack\(\) \(c net.Conn, r *bufio.Reader\)](#)

[func \(sc *ServerConn\) Pending\(\) int](#)

[func \(sc *ServerConn\) Read\(\) \(req *http.Request, err error\)](#)

[func \(sc *ServerConn\) Write\(req *http.Request, resp *http.Response\) error](#)

Package files

[chunked.go](#) [dump.go](#) [persist.go](#) [reverseproxy.go](#)

Variables

```
var (  
    ErrPersistEOF = &http.ProtocolError{ErrorString: "persistent con  
    ErrClosed      = &http.ProtocolError{ErrorString: "connection clo  
    ErrPipeline    = &http.ProtocolError{ErrorString: "pipeline error  
)  
  
var ErrLineTooLong = errors.New("header line too long")
```

func [DumpRequest](#)

```
func DumpRequest(req *http.Request, body bool) (dump []byte, err error)
```

DumpRequest returns the as-received wire representation of req, optionally including the request body, for debugging. DumpRequest is semantically a no-op, but in order to dump the body, it reads the body data into memory and changes req.Body to refer to the in-memory copy. The documentation for `http.Request.Write` details which fields of req are used.

func [DumpRequestOut](#)

```
func DumpRequestOut(req *http.Request, body bool) ([]byte, error)
```

DumpRequestOut is like DumpRequest but includes headers that the standard http.Transport adds, such as User-Agent.

func DumpResponse

```
func DumpResponse(resp *http.Response, body bool) (dump []byte, err
```

DumpResponse is like DumpRequest but dumps a response.

func [NewChunkedReader](#)

```
func NewChunkedReader(r io.Reader) io.Reader
```

NewChunkedReader returns a new chunkedReader that translates the data read from r out of HTTP "chunked" format before returning it. The chunkedReader returns io.EOF when the final 0-length chunk is read.

NewChunkedReader is not needed by normal applications. The http package automatically decodes chunking when reading response bodies.

func [NewChunkedWriter](#)

```
func NewChunkedWriter(w io.Writer) io.WriteCloser
```

NewChunkedWriter returns a new chunkedWriter that translates writes into HTTP "chunked" format before writing them to w. Closing the returned chunkedWriter sends the final 0-length chunk that marks the end of the stream.

NewChunkedWriter is not needed by normal applications. The http package adds chunking automatically if handlers don't set a Content-Length header. Using NewChunkedWriter inside a handler would result in double chunking or chunking with a Content-Length length, both of which are wrong.

type [ClientConn](#)

```
type ClientConn struct {  
    // contains filtered or unexported fields  
}
```

A ClientConn sends request and receives headers over an underlying connection, while respecting the HTTP keepalive logic. ClientConn supports hijacking the connection calling Hijack to regain control of the underlying net.Conn and deal with it as desired.

ClientConn is low-level and should not be needed by most applications. See Client.

func [NewClientConn](#)

```
func NewClientConn(c net.Conn, r *bufio.Reader) *ClientConn
```

NewClientConn returns a new ClientConn reading and writing c. If r is not nil, it is the buffer to use when reading c.

func [NewProxyClientConn](#)

```
func NewProxyClientConn(c net.Conn, r *bufio.Reader) *ClientConn
```

NewProxyClientConn works like NewClientConn but writes Requests using Request's WriteProxy method.

func (*ClientConn) [Close](#)

```
func (cc *ClientConn) Close() error
```

Close calls Hijack and then also closes the underlying connection

func (*ClientConn) [Do](#)

```
func (cc *ClientConn) Do(req *http.Request) (resp *http.Response, er
```

Do is convenience method that writes a request and reads a response.

func (*ClientConn) [Hijack](#)

```
func (cc *ClientConn) Hijack() (c net.Conn, r *bufio.Reader)
```

Hijack detaches the ClientConn and returns the underlying connection as well as the read-side bufio which may have some left over data. Hijack may be called before the user or Read have signaled the end of the keep-alive logic. The user should not call Hijack while Read or Write is in progress.

func (*ClientConn) [Pending](#)

```
func (cc *ClientConn) Pending() int
```

Pending returns the number of unanswered requests that have been sent on the connection.

func (*ClientConn) [Read](#)

```
func (cc *ClientConn) Read(req *http.Request) (resp *http.Response,
```

Read reads the next response from the wire. A valid response might be returned together with an ErrPersistEOF, which means that the remote requested that this be the last request serviced. Read can be called concurrently with Write, but not with another Read.

func (*ClientConn) [Write](#)

```
func (cc *ClientConn) Write(req *http.Request) (err error)
```

Write writes a request. An ErrPersistEOF error is returned if the connection has been closed in an HTTP keepalive sense. If req.Close equals true, the keepalive connection is logically closed after this request and the opposing server is informed. An ErrUnexpectedEOF indicates the remote closed the underlying TCP connection, which is usually considered as graceful close.

type [ReverseProxy](#)

```
type ReverseProxy struct {
    // Director must be a function which modifies
    // the request into a new request to be sent
    // using Transport. Its response is then copied
    // back to the original client unmodified.
    Director func(*http.Request)

    // The transport used to perform proxy requests.
    // If nil, http.DefaultTransport is used.
    Transport http.RoundTripper

    // FlushInterval specifies the flush interval
    // to flush to the client while copying the
    // response body.
    // If zero, no periodic flushing is done.
    FlushInterval time.Duration
}
```

ReverseProxy is an HTTP Handler that takes an incoming request and sends it to another server, proxying the response back to the client.

func [NewSingleHostReverseProxy](#)

```
func NewSingleHostReverseProxy(target *url.URL) *ReverseProxy
```

NewSingleHostReverseProxy returns a new ReverseProxy that rewrites URLs to the scheme, host, and base path provided in target. If the target's path is "/base" and the incoming request was for "/dir", the target request will be for /base/dir.

func (***ReverseProxy**) [ServeHTTP](#)

```
func (p *ReverseProxy) ServeHTTP(rw http.ResponseWriter, req *http.R
```

type [ServerConn](#)

```
type ServerConn struct {  
    // contains filtered or unexported fields  
}
```

A `ServerConn` reads requests and sends responses over an underlying connection, until the HTTP keepalive logic commands an end. `ServerConn` also allows hijacking the underlying connection by calling `Hijack` to regain control over the connection. `ServerConn` supports pipe-lining, i.e. requests can be read out of sync (but in the same order) while the respective responses are sent.

`ServerConn` is low-level and should not be needed by most applications. See `Server`.

func [NewServerConn](#)

```
func NewServerConn(c net.Conn, r *bufio.Reader) *ServerConn
```

`NewServerConn` returns a new `ServerConn` reading and writing `c`. If `r` is not nil, it is the buffer to use when reading `c`.

func (*ServerConn) [Close](#)

```
func (sc *ServerConn) Close() error
```

`Close` calls `Hijack` and then also closes the underlying connection

func (*ServerConn) [Hijack](#)

```
func (sc *ServerConn) Hijack() (c net.Conn, r *bufio.Reader)
```

`Hijack` detaches the `ServerConn` and returns the underlying connection as well as the read-side `bufio` which may have some left over data. `Hijack` may be called before `Read` has signaled the end of the keep-alive logic. The user should not call `Hijack` while `Read` or `Write` is in progress.

func (*ServerConn) [Pending](#)

```
func (sc *ServerConn) Pending() int
```

Pending returns the number of unanswered requests that have been received on the connection.

func (*ServerConn) [Read](#)

```
func (sc *ServerConn) Read() (req *http.Request, err error)
```

Read returns the next request on the wire. An ErrPersistEOF is returned if it is gracefully determined that there are no more requests (e.g. after the first request on an HTTP/1.0 connection, or after a Connection:close on a HTTP/1.1 connection).

func (*ServerConn) [Write](#)

```
func (sc *ServerConn) Write(req *http.Request, resp *http.Response)
```

Write writes resp in response to req. To close the connection gracefully, set the Response.Close field to true. Write should be considered operational until it returns an error, regardless of any errors returned on the Read side.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package pprof

```
import "net/http/pprof"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package pprof serves via its HTTP server runtime profiling data in the format expected by the pprof visualization tool. For more information about pprof, see <http://code.google.com/p/google-perftools/>.

The package is typically only imported for the side effect of registering its HTTP handlers. The handled paths all begin with /debug/pprof/.

To use pprof, link this package into your program:

```
import _ "net/http/pprof"
```

Then use the pprof tool to look at the heap profile:

```
go tool pprof http://localhost:6060/debug/pprof/heap
```

Or to look at a 30-second CPU profile:

```
go tool pprof http://localhost:6060/debug/pprof/profile
```

Or to view all available profiles:

```
go tool pprof http://localhost:6060/debug/pprof/
```

For a study of the facility in action, visit

<http://blog.golang.org/2011/06/profiling-go-programs.html>

Index

[func Cmdline\(w http.ResponseWriter, r *http.Request\)](#)

[func Handler\(name string\) http.Handler](#)

[func Index\(w http.ResponseWriter, r *http.Request\)](#)

[func Profile\(w http.ResponseWriter, r *http.Request\)](#)

[func Symbol\(w http.ResponseWriter, r *http.Request\)](#)

Package files

[pprof.go](#)

func [Cmdline](#)

```
func Cmdline(w http.ResponseWriter, r *http.Request)
```

Cmdline responds with the running program's command line, with arguments separated by NUL bytes. The package initialization registers it as `/debug/pprof/cmdline`.

func [Handler](#)

```
func Handler(name string) http.Handler
```

Handler returns an HTTP handler that serves the named profile.

func [Index](#)

```
func Index(w http.ResponseWriter, r *http.Request)
```

Index responds with the pprof-formatted profile named by the request. For example, `"/debug/pprof/heap"` serves the "heap" profile. Index responds to a request for `"/debug/pprof/"` with an HTML page listing the available profiles.

func [Profile](#)

```
func Profile(w http.ResponseWriter, r *http.Request)
```

Profile responds with the pprof-formatted cpu profile. The package initialization registers it as `/debug/pprof/profile`.

func [Symbol](#)

```
func Symbol(w http.ResponseWriter, r *http.Request)
```

Symbol looks up the program counters listed in the request, responding with a table mapping program counters to function names. The package initialization registers it as `/debug/pprof/symbol`.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package mail

```
import "net/mail"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package mail implements parsing of mail messages.

For the most part, this package follows the syntax as specified by RFC 5322.

Notable divergences:

- * Obsolete address formats are not parsed, including addresses with embedded route information.
- * Group addresses are not parsed.
- * The full range of spacing (the CFWS syntax element) is not support such as breaking addresses across lines.

Index

Variables

type Address

func (a *Address) String() string

type Header

func (h Header) AddressList(key string) ([]*Address, error)

func (h Header) Date() (time.Time, error)

func (h Header) Get(key string) string

type Message

func ReadMessage(r io.Reader) (msg *Message, err error)

Package files

[message.go](#)

Variables

```
var ErrHeaderNotPresent = errors.New("mail: header not in message")
```

type [Address](#)

```
type Address struct {  
    Name    string // Proper name; may be empty.  
    Address string // user@domain  
}
```

Address represents a single mail address. An address such as "Barry Gibbs <bg@example.com>" is represented as `Address{Name: "Barry Gibbs", Address: "bg@example.com"}`.

func (*Address) [String](#)

```
func (a *Address) String() string
```

String formats the address as a valid RFC 5322 address. If the address's name contains non-ASCII characters the name will be rendered according to RFC 2047.

type [Header](#)

```
type Header map[string][]string
```

A Header represents the key-value pairs in a mail message header.

func (Header) [AddressList](#)

```
func (h Header) AddressList(key string) ([]*Address, error)
```

AddressList parses the named header field as a list of addresses.

func (Header) [Date](#)

```
func (h Header) Date() (time.Time, error)
```

Date parses the Date header field.

func (Header) [Get](#)

```
func (h Header) Get(key string) string
```

Get gets the first value associated with the given key. If there are no values associated with the key, Get returns "".

type [Message](#)

```
type Message struct {  
    Header Header  
    Body   io.Reader  
}
```

A Message represents a parsed mail message.

func [ReadMessage](#)

```
func ReadMessage(r io.Reader) (msg *Message, err error)
```

ReadMessage reads a message from r. The headers are parsed, and the body of the message will be reading from r.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package rpc

```
import "net/rpc"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package `rpc` provides access to the exported methods of an object across a network or other I/O connection. A server registers an object, making it visible as a service with the name of the type of the object. After registration, exported methods of the object will be accessible remotely. A server may register multiple objects (services) of different types but it is an error to register multiple objects of the same type.

Only methods that satisfy these criteria will be made available for remote access; other methods will be ignored:

- the method is exported.
- the method has two arguments, both exported (or builtin) types.
- the method's second argument is a pointer.
- the method has return type `error`.

In effect, the method must look schematically like

```
func (t *T) MethodName(argType T1, replyType *T2) error
```

where `T`, `T1` and `T2` can be marshaled by encoding/gob. These requirements apply even if a different codec is used. (In future, these requirements may soften for custom codecs.)

The method's first argument represents the arguments provided by the caller; the second argument represents the result parameters to be returned to the caller. The method's return value, if non-nil, is passed back as a string that the client sees as if created by `errors.New`.

The server may handle requests on a single connection by calling `ServeConn`. More typically it will create a network listener and call `Accept` or, for an HTTP listener, `HandleHTTP` and `http.Serve`.

A client wishing to use the service establishes a connection and then invokes `NewClient` on the connection. The convenience function `Dial` (`DialHTTP`) performs both steps for a raw network connection (an HTTP connection). The resulting `Client` object has two methods, `Call` and `Go`, that specify the service and method to call, a pointer containing the arguments, and a pointer to receive

the result parameters.

The Call method waits for the remote call to complete while the Go method launches the call asynchronously and signals completion using the Call structure's Done channel.

Unless an explicit codec is set up, package encoding/gob is used to transport the data.

Here is a simple example. A server wishes to export an object of type Arith:

```
package server

type Args struct {
    A, B int
}

type Quotient struct {
    Quo, Rem int
}

type Arith int

func (t *Arith) Multiply(args *Args, reply *int) error {
    *reply = args.A * args.B
    return nil
}

func (t *Arith) Divide(args *Args, quo *Quotient) error {
    if args.B == 0 {
        return errors.New("divide by zero")
    }
    quo.Quo = args.A / args.B
    quo.Rem = args.A % args.B
    return nil
}
```

The server calls (for HTTP service):

```
arith := new(Arith)
rpc.Register(arith)
rpc.HandleHTTP()
l, e := net.Listen("tcp", ":1234")
if e != nil {
    log.Fatal("listen error:", e)
}
go http.Serve(l, nil)
```

At this point, clients can see a service "Arith" with methods "Arith.Multiply" and "Arith.Divide". To invoke one, a client first dials the server:

```
client, err := rpc.DialHTTP("tcp", serverAddress + ":1234")
if err != nil {
    log.Fatal("dialing:", err)
}
```

Then it can make a remote call:

```
// Synchronous call
args := &server.Args{7,8}
var reply int
err = client.Call("Arith.Multiply", args, &reply)
if err != nil {
    log.Fatal("arith error:", err)
}
fmt.Printf("Arith: %d*%d=%d", args.A, args.B, reply)
```

or

```
// Asynchronous call
quotient := new(Quotient)
divCall := client.Go("Arith.Divide", args, &quotient, nil)
replyCall := <-divCall.Done // will be equal to divCall
// check errors, print, etc.
```

A server implementation will often provide a simple, type-safe wrapper for the client.

Index

[Constants](#)

[Variables](#)

[func Accept\(lis net.Listener\)](#)

[func HandleHTTP\(\)](#)

[func Register\(rcvr interface{}\) error](#)

[func RegisterName\(name string, rcvr interface{}\) error](#)

[func ServeCodec\(codec ServerCodec\)](#)

[func ServeConn\(conn io.ReadWriteCloser\)](#)

[func ServeRequest\(codec ServerCodec\) error](#)

[type Call](#)

[type Client](#)

[func Dial\(network, address string\) \(*Client, error\)](#)

[func DialHTTP\(network, address string\) \(*Client, error\)](#)

[func DialHTTPPath\(network, address, path string\) \(*Client, error\)](#)

[func NewClient\(conn io.ReadWriteCloser\) *Client](#)

[func NewClientWithCodec\(codec ClientCodec\) *Client](#)

[func \(client *Client\) Call\(serviceMethod string, args interface{}, reply](#)

[interface{}\) error](#)

[func \(client *Client\) Close\(\) error](#)

[func \(client *Client\) Go\(serviceMethod string, args interface{}, reply](#)

[interface{}, done chan *Call\) *Call](#)

[type ClientCodec](#)

[type Request](#)

[type Response](#)

[type Server](#)

[func NewServer\(\) *Server](#)

[func \(server *Server\) Accept\(lis net.Listener\)](#)

[func \(server *Server\) HandleHTTP\(rpcPath, debugPath string\)](#)

[func \(server *Server\) Register\(rcvr interface{}\) error](#)

[func \(server *Server\) RegisterName\(name string, rcvr interface{}\) error](#)

[func \(server *Server\) ServeCodec\(codec ServerCodec\)](#)

[func \(server *Server\) ServeConn\(conn io.ReadWriteCloser\)](#)

[func \(server *Server\) ServeHTTP\(w http.ResponseWriter, req](#)

[*http.Request\)](#)

[func \(server *Server\) ServeRequest\(codec ServerCodec\) error](#)

```
type ServerCodec  
type ServerError  
func \(e ServerError\) Error\(\) string
```

Package files

[client.go](#) [debug.go](#) [server.go](#)

Constants

```
const (  
    // Defaults used by HandleHTTP  
    DefaultRPCPath    = "/_goRPC_"  
    DefaultDebugPath = "/debug/rpc"  
)
```

Variables

```
var DefaultServer = NewServer()
```

DefaultServer is the default instance of *Server.

```
var ErrShutdown = errors.New("connection is shut down")
```

func [Accept](#)

```
func Accept(lis net.Listener)
```

Accept accepts connections on the listener and serves requests to DefaultServer for each incoming connection. Accept blocks; the caller typically invokes it in a go statement.

func [HandleHTTP](#)

```
func HandleHTTP()
```

HandleHTTP registers an HTTP handler for RPC messages to DefaultServer on DefaultRPCPath and a debugging handler on DefaultDebugPath. It is still necessary to invoke `http.Serve()`, typically in a `go` statement.

func Register

```
func Register(rcvr interface{}) error
```

Register publishes the receiver's methods in the DefaultServer.

func RegisterName

```
func RegisterName(name string, rcvr interface{}) error
```

RegisterName is like Register but uses the provided name for the type instead of the receiver's concrete type.

func [ServeCodec](#)

```
func ServeCodec(codec ServerCodec)
```

ServeCodec is like ServeConn but uses the specified codec to decode requests and encode responses.

func [ServeConn](#)

```
func ServeConn(conn io.ReadWriteCloser)
```

ServeConn runs the DefaultServer on a single connection. ServeConn blocks, serving the connection until the client hangs up. The caller typically invokes ServeConn in a go statement. ServeConn uses the gob wire format (see package gob) on the connection. To use an alternate codec, use ServeCodec.

func [ServeRequest](#)

```
func ServeRequest(codec ServerCodec) error
```

ServeRequest is like ServeCodec but synchronously serves a single request. It does not close the codec upon completion.

type [Call](#)

```
type Call struct {
    ServiceMethod string // The name of the service and method
    Args          interface{} // The argument to the function (*stru
    Reply         interface{} // The reply from the function (*struc
    Error         error      // After completion, the error status.
    Done         chan *Call // Strobes when call is complete.
}
```

Call represents an active RPC.

type [Client](#)

```
type Client struct {  
    // contains filtered or unexported fields  
}
```

Client represents an RPC Client. There may be multiple outstanding Calls associated with a single Client, and a Client may be used by multiple goroutines simultaneously.

func [Dial](#)

```
func Dial(network, address string) (*Client, error)
```

Dial connects to an RPC server at the specified network address.

func [DialHTTP](#)

```
func DialHTTP(network, address string) (*Client, error)
```

DialHTTP connects to an HTTP RPC server at the specified network address listening on the default HTTP RPC path.

func [DialHTTPPath](#)

```
func DialHTTPPath(network, address, path string) (*Client, error)
```

DialHTTPPath connects to an HTTP RPC server at the specified network address and path.

func [NewClient](#)

```
func NewClient(conn io.ReadWriteCloser) *Client
```

NewClient returns a new Client to handle requests to the set of services at the other end of the connection. It adds a buffer to the write side of the connection so the header and payload are sent as a unit.

func [NewClientWithCodec](#)

```
func NewClientWithCodec(codec ClientCodec) *Client
```

NewClientWithCodec is like NewClient but uses the specified codec to encode requests and decode responses.

func (*Client) [Call](#)

```
func (client *Client) Call(serviceMethod string, args interface{}, r
```

Call invokes the named function, waits for it to complete, and returns its error status.

func (*Client) [Close](#)

```
func (client *Client) Close() error
```

func (*Client) [Go](#)

```
func (client *Client) Go(serviceMethod string, args interface{}, rep
```

Go invokes the function asynchronously. It returns the Call structure representing the invocation. The done channel will signal when the call is complete by returning the same Call object. If done is nil, Go will allocate a new channel. If non-nil, done must be buffered or Go will deliberately crash.

type [ClientCodec](#)

```
type ClientCodec interface {
    WriteRequest(*Request, interface{}) error
    ReadResponseHeader(*Response) error
    ReadResponseBody(interface{}) error

    Close() error
}
```

A ClientCodec implements writing of RPC requests and reading of RPC responses for the client side of an RPC session. The client calls WriteRequest to write a request to the connection and calls ReadResponseHeader and ReadResponseBody in pairs to read responses. The client calls Close when finished with the connection. ReadResponseBody may be called with a nil argument to force the body of the response to be read and then discarded.

type [Request](#)

```
type Request struct {  
    ServiceMethod string // format: "Service.Method"  
    Seq           uint64 // sequence number chosen by client  
    // contains filtered or unexported fields  
}
```

Request is a header written before every RPC call. It is used internally but documented here as an aid to debugging, such as when analyzing network traffic.

type [Response](#)

```
type Response struct {
    ServiceMethod string // echoes that of the Request
    Seq           uint64 // echoes that of the request
    Error         string // error, if any.
    // contains filtered or unexported fields
}
```

Response is a header written before every RPC return. It is used internally but documented here as an aid to debugging, such as when analyzing network traffic.

type [Server](#)

```
type Server struct {  
    // contains filtered or unexported fields  
}
```

Server represents an RPC Server.

func [NewServer](#)

```
func NewServer() *Server
```

NewServer returns a new Server.

func (*Server) [Accept](#)

```
func (server *Server) Accept(lis net.Listener)
```

Accept accepts connections on the listener and serves requests for each incoming connection. Accept blocks; the caller typically invokes it in a go statement.

func (*Server) [HandleHTTP](#)

```
func (server *Server) HandleHTTP(rpcPath, debugPath string)
```

HandleHTTP registers an HTTP handler for RPC messages on rpcPath, and a debugging handler on debugPath. It is still necessary to invoke http.Serve(), typically in a go statement.

func (*Server) [Register](#)

```
func (server *Server) Register(rcvr interface{}) error
```

Register publishes in the server the set of methods of the receiver value that satisfy the following conditions:

- exported method
- two arguments, both pointers to exported structs
- one return value, of type error

It returns an error if the receiver is not an exported type or has no suitable methods. The client accesses each method using a string of the form "Type.Method", where Type is the receiver's concrete type.

func (*Server) [RegisterName](#)

```
func (server *Server) RegisterName(name string, rcvr interface{}) error
```

RegisterName is like Register but uses the provided name for the type instead of the receiver's concrete type.

func (*Server) [ServeCodec](#)

```
func (server *Server) ServeCodec(codec ServerCodec)
```

ServeCodec is like ServeConn but uses the specified codec to decode requests and encode responses.

func (*Server) [ServeConn](#)

```
func (server *Server) ServeConn(conn io.ReadWriteCloser)
```

ServeConn runs the server on a single connection. ServeConn blocks, serving the connection until the client hangs up. The caller typically invokes ServeConn in a go statement. ServeConn uses the gob wire format (see package gob) on the connection. To use an alternate codec, use ServeCodec.

func (*Server) [ServeHTTP](#)

```
func (server *Server) ServeHTTP(w http.ResponseWriter, req *http.Request)
```

ServeHTTP implements an http.Handler that answers RPC requests.

func (*Server) [ServeRequest](#)

```
func (server *Server) ServeRequest(codec ServerCodec) error
```

ServeRequest is like ServeCodec but synchronously serves a single request. It does not close the codec upon completion.

type [ServerCodec](#)

```
type ServerCodec interface {  
    ReadRequestHeader(*Request) error  
    ReadRequestBody(interface{}) error  
    WriteResponse(*Response, interface{}) error  
  
    Close() error  
}
```

A `ServerCodec` implements reading of RPC requests and writing of RPC responses for the server side of an RPC session. The server calls `ReadRequestHeader` and `ReadRequestBody` in pairs to read requests from the connection, and it calls `WriteResponse` to write a response back. The server calls `Close` when finished with the connection. `ReadRequestBody` may be called with a `nil` argument to force the body of the request to be read and discarded.

type [ServerError](#)

```
type ServerError string
```

ServerError represents an error that has been returned from the remote side of the RPC connection.

func (ServerError) [Error](#)

```
func (e ServerError) Error() string
```

Subdirectories

Name	Synopsis
jsonrpc	Package jsonrpc implements a JSON-RPC ClientCodec and ServerCodec for the rpc package.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package jsonrpc

```
import "net/rpc/jsonrpc"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package jsonrpc implements a JSON-RPC ClientCodec and ServerCodec for the rpc package.

Index

[func Dial\(network, address string\) \(*rpc.Client, error\)](#)

[func NewClient\(conn io.ReadWriteCloser\) *rpc.Client](#)

[func NewClientCodec\(conn io.ReadWriteCloser\) rpc.ClientCodec](#)

[func NewServerCodec\(conn io.ReadWriteCloser\) rpc.ServerCodec](#)

[func ServeConn\(conn io.ReadWriteCloser\)](#)

Package files

[client.go](#) [server.go](#)

func [Dial](#)

```
func Dial(network, address string) (*rpc.Client, error)
```

Dial connects to a JSON-RPC server at the specified network address.

func [NewClient](#)

```
func NewClient(conn io.ReadWriteCloser) *rpc.Client
```

NewClient returns a new `rpc.Client` to handle requests to the set of services at the other end of the connection.

func [NewClientCodec](#)

```
func NewClientCodec(conn io.ReadWriteCloser) rpc.ClientCodec
```

NewClientCodec returns a new rpc.ClientCodec using JSON-RPC on conn.

func [NewServerCodec](#)

```
func NewServerCodec(conn io.ReadWriteCloser) rpc.ServerCodec
```

NewServerCodec returns a new rpc.ServerCodec using JSON-RPC on conn.

func [ServeConn](#)

```
func ServeConn(conn io.ReadWriteCloser)
```

ServeConn runs the JSON-RPC server on a single connection. ServeConn blocks, serving the connection until the client hangs up. The caller typically invokes ServeConn in a go statement.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package smtp

```
import "net/smtp"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package smtp implements the Simple Mail Transfer Protocol as defined in RFC 5321. It also implements the following extensions:

8BITMIME	RFC 1652
AUTH	RFC 2554
STARTTLS	RFC 3207

Additional extensions may be handled by clients.

Index

[func SendMail\(addr string, a Auth, from string, to \[\]string, msg \[\]byte\)](#)

[error](#)

[type Auth](#)

[func CRAMMD5Auth\(username, secret string\) Auth](#)

[func PlainAuth\(identity, username, password, host string\) Auth](#)

[type Client](#)

[func Dial\(addr string\) \(*Client, error\)](#)

[func NewClient\(conn net.Conn, host string\) \(*Client, error\)](#)

[func \(c *Client\) Auth\(a Auth\) error](#)

[func \(c *Client\) Data\(\) \(io.WriteCloser, error\)](#)

[func \(c *Client\) Extension\(ext string\) \(bool, string\)](#)

[func \(c *Client\) Mail\(from string\) error](#)

[func \(c *Client\) Quit\(\) error](#)

[func \(c *Client\) Rcpt\(to string\) error](#)

[func \(c *Client\) Reset\(\) error](#)

[func \(c *Client\) StartTLS\(config *tls.Config\) error](#)

[func \(c *Client\) Verify\(addr string\) error](#)

[type ServerInfo](#)

Package files

[auth.go smtp.go](#)

func [SendMail](#)

```
func SendMail(addr string, a Auth, from string, to []string, msg []b
```

SendMail connects to the server at addr, switches to TLS if possible, authenticates with mechanism a if possible, and then sends an email from address from, to addresses to, with message msg.

type [Auth](#)

```
type Auth interface {
    // Start begins an authentication with a server.
    // It returns the name of the authentication protocol
    // and optionally data to include in the initial AUTH message
    // sent to the server. It can return proto == "" to indicate
    // that the authentication should be skipped.
    // If it returns a non-nil error, the SMTP client aborts
    // the authentication attempt and closes the connection.
    Start(server *ServerInfo) (proto string, toServer []byte, err error)

    // Next continues the authentication. The server has just sent
    // the fromServer data. If more is true, the server expects a
    // response, which Next should return as toServer; otherwise
    // Next should return toServer == nil.
    // If Next returns a non-nil error, the SMTP client aborts
    // the authentication attempt and closes the connection.
    Next(fromServer []byte, more bool) (toServer []byte, err error)
}
```

Auth is implemented by an SMTP authentication mechanism.

func [CRAMMD5Auth](#)

```
func CRAMMD5Auth(username, secret string) Auth
```

CRAMMD5Auth returns an Auth that implements the CRAM-MD5 authentication mechanism as defined in RFC 2195. The returned Auth uses the given username and secret to authenticate to the server using the challenge-response mechanism.

func [PlainAuth](#)

```
func PlainAuth(identity, username, password, host string) Auth
```

PlainAuth returns an Auth that implements the PLAIN authentication mechanism as defined in RFC 4616. The returned Auth uses the given username and password to authenticate on TLS connections to host and act as identity. Usually identity will be left blank to act as username.

type [Client](#)

```
type Client struct {  
    // Text is the textproto.Conn used by the Client. It is exported  
    // clients to add extensions.  
    Text *textproto.Conn  
    // contains filtered or unexported fields  
}
```

A Client represents a client connection to an SMTP server.

func [Dial](#)

```
func Dial(addr string) (*Client, error)
```

Dial returns a new Client connected to an SMTP server at addr.

func [NewClient](#)

```
func NewClient(conn net.Conn, host string) (*Client, error)
```

NewClient returns a new Client using an existing connection and host as a server name to be used when authenticating.

func (***Client**) [Auth](#)

```
func (c *Client) Auth(a Auth) error
```

Auth authenticates a client using the provided authentication mechanism. A failed authentication closes the connection. Only servers that advertise the AUTH extension support this function.

func (***Client**) [Data](#)

```
func (c *Client) Data() (io.WriteCloser, error)
```

Data issues a DATA command to the server and returns a writer that can be used to write the data. The caller should close the writer before calling any more methods on c. A call to Data must be preceded by one or more calls to Rcpt.

func (*Client) [Extension](#)

```
func (c *Client) Extension(ext string) (bool, string)
```

Extension reports whether an extension is supported by the server. The extension name is case-insensitive. If the extension is supported, Extension also returns a string that contains any parameters the server specifies for the extension.

func (*Client) [Mail](#)

```
func (c *Client) Mail(from string) error
```

Mail issues a MAIL command to the server using the provided email address. If the server supports the 8BITMIME extension, Mail adds the BODY=8BITMIME parameter. This initiates a mail transaction and is followed by one or more Rcpt calls.

func (*Client) [Quit](#)

```
func (c *Client) Quit() error
```

Quit sends the QUIT command and closes the connection to the server.

func (*Client) [Rcpt](#)

```
func (c *Client) Rcpt(to string) error
```

Rcpt issues a RCPT command to the server using the provided email address. A call to Rcpt must be preceded by a call to Mail and may be followed by a Data call or another Rcpt call.

func (*Client) [Reset](#)

```
func (c *Client) Reset() error
```

Reset sends the RSET command to the server, aborting the current mail transaction.

func (*Client) [StartTLS](#)

```
func (c *Client) StartTLS(config *tls.Config) error
```

StartTLS sends the STARTTLS command and encrypts all further communication. Only servers that advertise the STARTTLS extension support this function.

func (*Client) [Verify](#)

```
func (c *Client) Verify(addr string) error
```

Verify checks the validity of an email address on the server. If Verify returns nil, the address is valid. A non-nil return does not necessarily indicate an invalid address. Many servers will not verify addresses for security reasons.

type [ServerInfo](#)

```
type ServerInfo struct {  
    Name string    // SMTP server name  
    TLS  bool       // using TLS, with valid certificate for Name  
    Auth []string  // advertised authentication mechanisms  
}
```

ServerInfo records information about an SMTP server.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package textproto

```
import "net/textproto"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package textproto implements generic support for text-based request/response protocols in the style of HTTP, NNTP, and SMTP.

The package provides:

Error, which represents a numeric error response from a server.

Pipeline, to manage pipelined requests and responses in a client.

Reader, to read numeric response code lines, key: value headers, lines wrapped with leading spaces on continuation lines, and whole text blocks ending with a dot on a line by itself.

Writer, to write dot-encoded text blocks.

Conn, a convenient packaging of Reader, Writer, and Pipeline for use with a single network connection.

Index

[func CanonicalMIMEHeaderKey\(s string\) string](#)
[type Conn](#)
 [func Dial\(network, addr string\) \(*Conn, error\)](#)
 [func NewConn\(conn io.ReadWriter\) *Conn](#)
 [func \(c *Conn\) Close\(\) error](#)
 [func \(c *Conn\) Cmd\(format string, args ...interface{}\) \(id uint, err error\)](#)
[type Error](#)
 [func \(e *Error\) Error\(\) string](#)
[type MIMEHeader](#)
 [func \(h MIMEHeader\) Add\(key, value string\)](#)
 [func \(h MIMEHeader\) Del\(key string\)](#)
 [func \(h MIMEHeader\) Get\(key string\) string](#)
 [func \(h MIMEHeader\) Set\(key, value string\)](#)
[type Pipeline](#)
 [func \(p *Pipeline\) EndRequest\(id uint\)](#)
 [func \(p *Pipeline\) EndResponse\(id uint\)](#)
 [func \(p *Pipeline\) Next\(\) uint](#)
 [func \(p *Pipeline\) StartRequest\(id uint\)](#)
 [func \(p *Pipeline\) StartResponse\(id uint\)](#)
[type ProtocolError](#)
 [func \(p ProtocolError\) Error\(\) string](#)
[type Reader](#)
 [func NewReader\(r *bufio.Reader\) *Reader](#)
 [func \(r *Reader\) DotReader\(\) io.Reader](#)
 [func \(r *Reader\) ReadCodeLine\(expectCode int\) \(code int, message string, err error\)](#)
 [func \(r *Reader\) ReadContinuedLine\(\) \(string, error\)](#)
 [func \(r *Reader\) ReadContinuedLineBytes\(\) \(\[\]byte, error\)](#)
 [func \(r *Reader\) ReadDotBytes\(\) \(\[\]byte, error\)](#)
 [func \(r *Reader\) ReadDotLines\(\) \(\[\]string, error\)](#)
 [func \(r *Reader\) ReadLine\(\) \(string, error\)](#)
 [func \(r *Reader\) ReadLineBytes\(\) \(\[\]byte, error\)](#)
 [func \(r *Reader\) ReadMIMEHeader\(\) \(MIMEHeader, error\)](#)
 [func \(r *Reader\) ReadResponse\(expectCode int\) \(code int, message string, err error\)](#)

[type Writer](#)

[func NewWriter\(w *bufio.Writer\) *Writer](#)

[func \(w *Writer\) DotWriter\(\) io.WriteCloser](#)

[func \(w *Writer\) PrintfLine\(format string, args ...interface{}\) error](#)

[Bugs](#)

Package files

[header.go](#) [pipeline.go](#) [reader.go](#) [textproto.go](#) [writer.go](#)

func [CanonicalMIMEHeaderKey](#)

```
func CanonicalMIMEHeaderKey(s string) string
```

CanonicalMIMEHeaderKey returns the canonical format of the MIME header key s. The canonicalization converts the first letter and any letter following a hyphen to upper case; the rest are converted to lowercase. For example, the canonical key for "accept-encoding" is "Accept-Encoding".

type [Conn](#)

```
type Conn struct {
    Reader
    Writer
    Pipeline
    // contains filtered or unexported fields
}
```

A Conn represents a textual network protocol connection. It consists of a Reader and Writer to manage I/O and a Pipeline to sequence concurrent requests on the connection. These embedded types carry methods with them; see the documentation of those types for details.

func [Dial](#)

```
func Dial(network, addr string) (*Conn, error)
```

Dial connects to the given address on the given network using net.Dial and then returns a new Conn for the connection.

func [NewConn](#)

```
func NewConn(conn io.ReadWriteCloser) *Conn
```

NewConn returns a new Conn using conn for I/O.

func (***Conn**) [Close](#)

```
func (c *Conn) Close() error
```

Close closes the connection.

func (***Conn**) [Cmd](#)

```
func (c *Conn) Cmd(format string, args ...interface{}) (id uint, err
```

Cmd is a convenience method that sends a command after waiting its turn in the pipeline. The command text is the result of formatting format with args and

appending `\r\n`. `Cmd` returns the id of the command, for use with `StartResponse` and `EndResponse`.

For example, a client might run a `HELP` command that returns a dot-body by using:

```
id, err := c.Cmd("HELP")
if err != nil {
    return nil, err
}

c.StartResponse(id)
defer c.EndResponse(id)

if _, _, err = c.ReadCodeLine(110); err != nil {
    return nil, err
}
text, err := c.ReadDotAll()
if err != nil {
    return nil, err
}
return c.ReadCodeLine(250)
```

type [Error](#)

```
type Error struct {  
    Code int  
    Msg  string  
}
```

An Error represents a numeric error response from a server.

func (*Error) [Error](#)

```
func (e *Error) Error() string
```

type [MIMEHeader](#)

```
type MIMEHeader map[string][]string
```

A MIMEHeader represents a MIME-style header mapping keys to sets of values.

func (MIMEHeader) [Add](#)

```
func (h MIMEHeader) Add(key, value string)
```

Add adds the key, value pair to the header. It appends to any existing values associated with key.

func (MIMEHeader) [Del](#)

```
func (h MIMEHeader) Del(key string)
```

Del deletes the values associated with key.

func (MIMEHeader) [Get](#)

```
func (h MIMEHeader) Get(key string) string
```

Get gets the first value associated with the given key. If there are no values associated with the key, Get returns "". Get is a convenience method. For more complex queries, access the map directly.

func (MIMEHeader) [Set](#)

```
func (h MIMEHeader) Set(key, value string)
```

Set sets the header entries associated with key to the single element value. It replaces any existing values associated with key.

type [Pipeline](#)

```
type Pipeline struct {  
    // contains filtered or unexported fields  
}
```

A Pipeline manages a pipelined in-order request/response sequence.

To use a Pipeline `p` to manage multiple clients on a connection, each client should run:

```
id := p.Next() // take a number  
  
p.StartRequest(id) // wait for turn to send request  
?send request?  
p.EndRequest(id) // notify Pipeline that request is sent  
  
p.StartResponse(id) // wait for turn to read response  
?read response?  
p.EndResponse(id) // notify Pipeline that response is read
```

A pipelined server can use the same calls to ensure that responses computed in parallel are written in the correct order.

func (*Pipeline) [EndRequest](#)

```
func (p *Pipeline) EndRequest(id uint)
```

`EndRequest` notifies `p` that the request with the given `id` has been sent (or, if this is a server, received).

func (*Pipeline) [EndResponse](#)

```
func (p *Pipeline) EndResponse(id uint)
```

`EndResponse` notifies `p` that the response with the given `id` has been received (or, if this is a server, sent).

func (*Pipeline) [Next](#)

```
func (p *Pipeline) Next() uint
```

Next returns the next id for a request/response pair.

func (*Pipeline) [StartRequest](#)

```
func (p *Pipeline) StartRequest(id uint)
```

StartRequest blocks until it is time to send (or, if this is a server, receive) the request with the given id.

func (*Pipeline) [StartResponse](#)

```
func (p *Pipeline) StartResponse(id uint)
```

StartResponse blocks until it is time to receive (or, if this is a server, send) the request with the given id.

type [ProtocolError](#)

```
type ProtocolError string
```

A ProtocolError describes a protocol violation such as an invalid response or a hung-up connection.

func (ProtocolError) [Error](#)

```
func (p ProtocolError) Error() string
```

type [Reader](#)

```
type Reader struct {  
    R *bufio.Reader  
    // contains filtered or unexported fields  
}
```

A Reader implements convenience methods for reading requests or responses from a text protocol network connection.

func [NewReader](#)

```
func NewReader(r *bufio.Reader) *Reader
```

NewReader returns a new Reader reading from r.

func (***Reader**) [DotReader](#)

```
func (r *Reader) DotReader() io.Reader
```

DotReader returns a new Reader that satisfies Reads using the decoded text of a dot-encoded block read from r. The returned Reader is only valid until the next call to a method on r.

Dot encoding is a common framing used for data blocks in text protocols such as SMTP. The data consists of a sequence of lines, each of which ends in "\r\n". The sequence itself ends at a line containing just a dot: ".\r\n". Lines beginning with a dot are escaped with an additional dot to avoid looking like the end of the sequence.

The decoded form returned by the Reader's Read method rewrites the "\r\n" line endings into the simpler "\n", removes leading dot escapes if present, and stops with error io.EOF after consuming (and discarding) the end-of-sequence line.

func (***Reader**) [ReadCodeLine](#)

```
func (r *Reader) ReadCodeLine(expectCode int) (code int, message str
```

ReadCodeLine reads a response code line of the form

code message

where code is a 3-digit status code and the message extends to the rest of the line. An example of such a line is:

```
220 plan9.bell-labs.com ESMTTP
```

If the prefix of the status does not match the digits in `expectCode`, `ReadCodeLine` returns with `err` set to `&Error{code, message}`. For example, if `expectCode` is 31, an error will be returned if the status is not in the range [310,319].

If the response is multi-line, `ReadCodeLine` returns an error.

An `expectCode <= 0` disables the check of the status code.

func (*Reader) [ReadContinuedLine](#)

```
func (r *Reader) ReadContinuedLine() (string, error)
```

`ReadContinuedLine` reads a possibly continued line from `r`, eliding the final trailing ASCII white space. Lines after the first are considered continuations if they begin with a space or tab character. In the returned data, continuation lines are separated from the previous line only by a single space: the newline and leading white space are removed.

For example, consider this input:

```
Line 1
  continued...
Line 2
```

The first call to `ReadContinuedLine` will return "Line 1 continued..." and the second will return "Line 2".

A line consisting of only white space is never continued.

func (*Reader) [ReadContinuedLineBytes](#)

```
func (r *Reader) ReadContinuedLineBytes() ([]byte, error)
```

ReadContinuedLineBytes is like ReadContinuedLine but returns a []byte instead of a string.

func (*Reader) [ReadDotBytes](#)

```
func (r *Reader) ReadDotBytes() ([]byte, error)
```

ReadDotBytes reads a dot-encoding and returns the decoded data.

See the documentation for the DotReader method for details about dot-encoding.

func (*Reader) [ReadDotLines](#)

```
func (r *Reader) ReadDotLines() ([]string, error)
```

ReadDotLines reads a dot-encoding and returns a slice containing the decoded lines, with the final \r\n or \n elided from each.

See the documentation for the DotReader method for details about dot-encoding.

func (*Reader) [ReadLine](#)

```
func (r *Reader) ReadLine() (string, error)
```

ReadLine reads a single line from r, eliding the final \n or \r\n from the returned string.

func (*Reader) [ReadLineBytes](#)

```
func (r *Reader) ReadLineBytes() ([]byte, error)
```

ReadLineBytes is like ReadLine but returns a []byte instead of a string.

func (*Reader) [ReadMIMEHeader](#)

```
func (r *Reader) ReadMIMEHeader() (MIMEHeader, error)
```

ReadMIMEHeader reads a MIME-style header from r. The header is a sequence of possibly continued Key: Value lines ending in a blank line. The returned map m maps CanonicalMIMEHeaderKey(key) to a sequence of values in the same

order encountered in the input.

For example, consider this input:

```
My-Key: Value 1
Long-Key: Even
         Longer Value
My-Key: Value 2
```

Given that input, `ReadMIMEHeader` returns the map:

```
map[string][]string{
    "My-Key": {"Value 1", "Value 2"},
    "Long-Key": {"Even Longer Value"},
}
```

func (*Reader) [ReadResponse](#)

```
func (r *Reader) ReadResponse(expectCode int) (code int, message string, err error)
```

`ReadResponse` reads a multi-line response of the form:

```
code-message line 1
code-message line 2
...
code message line n
```

where `code` is a 3-digit status code. The first line starts with the code and a hyphen. The response is terminated by a line that starts with the same code followed by a space. Each line in `message` is separated by a newline (`\n`).

See page 36 of RFC 959 (<http://www.ietf.org/rfc/rfc959.txt>) for details.

If the prefix of the status does not match the digits in `expectCode`, `ReadResponse` returns with `err` set to `&Error{code, message}`. For example, if `expectCode` is 31, an error will be returned if the status is not in the range [310,319].

An `expectCode` ≤ 0 disables the check of the status code.

type [Writer](#)

```
type Writer struct {
    w *bufio.Writer
    // contains filtered or unexported fields
}
```

A `Writer` implements convenience methods for writing requests or responses to a text protocol network connection.

func [NewWriter](#)

```
func NewWriter(w *bufio.Writer) *Writer
```

`NewWriter` returns a new `Writer` writing to `w`.

func (*Writer) [DotWriter](#)

```
func (w *Writer) DotWriter() io.WriteCloser
```

`DotWriter` returns a writer that can be used to write a dot-encoding to `w`. It takes care of inserting leading dots when necessary, translating line-ending `\n` into `\r\n`, and adding the final `.\r\n` line when the `DotWriter` is closed. The caller should close the `DotWriter` before the next call to a method on `w`.

See the documentation for `Reader`'s `DotReader` method for details about dot-encoding.

func (*Writer) [PrintfLine](#)

```
func (w *Writer) PrintfLine(format string, args ...interface{}) error
```

`PrintfLine` writes the formatted output followed by `\r\n`.

Bugs

To let callers manage exposure to denial of service attacks, Reader should allow them to set and reset a limit on the number of bytes read from the connection.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package url

```
import "net/url"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package url parses URLs and implements query escaping. See RFC 3986.

Index

[func QueryEscape\(s string\) string](#)
[func QueryUnescape\(s string\) \(string, error\)](#)
[type Error](#)
 [func \(e *Error\) Error\(\) string](#)
[type EscapeError](#)
 [func \(e EscapeError\) Error\(\) string](#)
[type URL](#)
 [func Parse\(rawurl string\) \(url *URL, err error\)](#)
 [func ParseRequestURI\(rawurl string\) \(url *URL, err error\)](#)
 [func \(u *URL\) IsAbs\(\) bool](#)
 [func \(u *URL\) Parse\(ref string\) \(*URL, error\)](#)
 [func \(u *URL\) Query\(\) Values](#)
 [func \(u *URL\) RequestURI\(\) string](#)
 [func \(u *URL\) ResolveReference\(ref *URL\) *URL](#)
 [func \(u *URL\) String\(\) string](#)
[type Userinfo](#)
 [func User\(username string\) *Userinfo](#)
 [func UserPassword\(username, password string\) *Userinfo](#)
 [func \(u *Userinfo\) Password\(\) \(string, bool\)](#)
 [func \(u *Userinfo\) String\(\) string](#)
 [func \(u *Userinfo\) Username\(\) string](#)
[type Values](#)
 [func ParseQuery\(query string\) \(m Values, err error\)](#)
 [func \(v Values\) Add\(key, value string\)](#)
 [func \(v Values\) Del\(key string\)](#)
 [func \(v Values\) Encode\(\) string](#)
 [func \(v Values\) Get\(key string\) string](#)
 [func \(v Values\) Set\(key, value string\)](#)

Examples

[URL](#)
[Values](#)

Package files

[url.go](#)

func [QueryEscape](#)

```
func QueryEscape(s string) string
```

QueryEscape escapes the string so it can be safely placed inside a URL query.

func [QueryUnescape](#)

```
func QueryUnescape(s string) (string, error)
```

QueryUnescape does the inverse transformation of QueryEscape, converting %AB into the byte 0xAB and '+' into ' ' (space). It returns an error if any % is not followed by two hexadecimal digits.

type [Error](#)

```
type Error struct {  
    Op string  
    URL string  
    Err error  
}
```

Error reports an error and the operation and URL that caused it.

func (***Error**) [Error](#)

```
func (e *Error) Error() string
```

type [EscapeError](#)

type EscapeError string

func (EscapeError) [Error](#)

func (e EscapeError) Error() string

type [URL](#)

```
type URL struct {
    Scheme    string
    Opaque    string    // encoded opaque data
    User      *UserInfo // username and password information
    Host      string
    Path      string
    RawQuery  string // encoded query values, without '?'
    Fragment  string // fragment for references, without '#'
}
```

A URL represents a parsed URL (technically, a URI reference). The general form represented is:

```
scheme://[userinfo@]host/path[?query][#fragment]
```

URLs that do not start with a slash after the scheme are interpreted as:

```
scheme:opaque[?query][#fragment]
```

? Example

? Example

Code:

```
u, err := url.Parse("http://bing.com/search?q=dotnet")
if err != nil {
    log.Fatal(err)
}
u.Scheme = "https"
u.Host = "google.com"
q := u.Query()
q.Set("q", "golang")
u.RawQuery = q.Encode()
fmt.Println(u)
```

Output:

```
https://google.com/search?q=golang
```

func [Parse](#)

```
func Parse(rawurl string) (url *URL, err error)
```

Parse parses rawurl into a URL structure. The rawurl may be relative or absolute.

func [ParseRequestURI](#)

```
func ParseRequestURI(rawurl string) (url *URL, err error)
```

ParseRequestURI parses rawurl into a URL structure. It assumes that rawurl was received in an HTTP request, so the rawurl is interpreted only as an absolute URI or an absolute path. The string rawurl is assumed not to have a #fragment suffix. (Web browsers strip #fragment before sending the URL to a web server.)

func (*URL) [IsAbs](#)

```
func (u *URL) IsAbs() bool
```

IsAbs returns true if the URL is absolute.

func (*URL) [Parse](#)

```
func (u *URL) Parse(ref string) (*URL, error)
```

Parse parses a URL in the context of the receiver. The provided URL may be relative or absolute. Parse returns nil, err on parse failure, otherwise its return value is the same as ResolveReference.

func (*URL) [Query](#)

```
func (u *URL) Query() Values
```

Query parses RawQuery and returns the corresponding values.

func (*URL) [RequestURI](#)

```
func (u *URL) RequestURI() string
```

RequestURI returns the encoded path?query or opaque?query string that would be used in an HTTP request for u.

func (*URL) [ResolveReference](#)

```
func (u *URL) ResolveReference(ref *URL) *URL
```

ResolveReference resolves a URI reference to an absolute URI from an absolute base URI, per RFC 2396 Section 5.2. The URI reference may be relative or absolute. ResolveReference always returns a new URL instance, even if the returned URL is identical to either the base or reference. If ref is an absolute URL, then ResolveReference ignores base and returns a copy of ref.

func (*URL) [String](#)

```
func (u *URL) String() string
```

String reassembles the URL into a valid URL string.

type [Userinfo](#)

```
type Userinfo struct {  
    // contains filtered or unexported fields  
}
```

The `Userinfo` type is an immutable encapsulation of username and password details for a URL. An existing `Userinfo` value is guaranteed to have a username set (potentially empty, as allowed by RFC 2396), and optionally a password.

func [User](#)

```
func User(username string) *Userinfo
```

`User` returns a `Userinfo` containing the provided username and no password set.

func [UserPassword](#)

```
func UserPassword(username, password string) *Userinfo
```

`UserPassword` returns a `Userinfo` containing the provided username and password. This functionality should only be used with legacy web sites. RFC 2396 warns that interpreting `Userinfo` this way “is NOT RECOMMENDED, because the passing of authentication information in clear text (such as URI) has proven to be a security risk in almost every case where it has been used.”

func (*Userinfo) [Password](#)

```
func (u *Userinfo) Password() (string, bool)
```

`Password` returns the password in case it is set, and whether it is set.

func (*Userinfo) [String](#)

```
func (u *Userinfo) String() string
```

`String` returns the encoded userinfo information in the standard form of "username[:password]".

func (*Userinfo) [Username](#)

```
func (u *Userinfo) Username() string
```

Username returns the username.

type [Values](#)

```
type Values map[string][]string
```

Values maps a string key to a list of values. It is typically used for query parameters and form values. Unlike in the `http.Header` map, the keys in a `Values` map are case-sensitive.

? Example

? Example

Code:

```
v := url.Values{}
v.Set("name", "Ava")
v.Add("friend", "Jess")
v.Add("friend", "Sarah")
v.Add("friend", "Zoe")
// v.Encode() == "name=Ava&friend=Jess&friend=Sarah&friend=Zoe"
fmt.Println(v.Get("name"))
fmt.Println(v.Get("friend"))
fmt.Println(v["friend"])
```

Output:

```
Ava
Jess
[Jess Sarah Zoe]
```

func [ParseQuery](#)

```
func ParseQuery(query string) (m Values, err error)
```

`ParseQuery` parses the URL-encoded query string and returns a map listing the values specified for each key. `ParseQuery` always returns a non-nil map containing all the valid query parameters found; `err` describes the first decoding error encountered, if any.

func (Values) [Add](#)

```
func (v Values) Add(key, value string)
```

Add adds the key to value. It appends to any existing values associated with key.

func (Values) [Del](#)

```
func (v Values) Del(key string)
```

Del deletes the values associated with key.

func (Values) [Encode](#)

```
func (v Values) Encode() string
```

Encode encodes the values into “URL encoded” form. e.g. "foo=bar&bar=baz"

func (Values) [Get](#)

```
func (v Values) Get(key string) string
```

Get gets the first value associated with the given key. If there are no values associated with the key, Get returns the empty string. To access multiple values, use the map directly.

func (Values) [Set](#)

```
func (v Values) Set(key, value string)
```

Set sets the key to value. It replaces any existing values.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package os

```
import "os"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package `os` provides a platform-independent interface to operating system functionality. The design is Unix-like, although the error handling is Go-like; failing calls return values of type `error` rather than error numbers. Often, more information is available within the error. For example, if a call that takes a file name fails, such as `Open` or `Stat`, the error will include the failing file name when printed and will be of type `*PathError`, which may be unpacked for more information.

The `os` interface is intended to be uniform across all operating systems. Features not generally available appear in the system-specific package `syscall`.

Here is a simple example, opening a file and reading some of it.

```
file, err := os.Open("file.go") // For read access.
if err != nil {
    log.Fatal(err)
}
```

If the open fails, the error string will be self-explanatory, like

```
open file.go: no such file or directory
```

The file's data can then be read into a slice of bytes. `Read` and `Write` take their byte counts from the length of the argument slice.

```
data := make([]byte, 100)
count, err := file.Read(data)
if err != nil {
    log.Fatal(err)
}
fmt.Printf("read %d bytes: %q\n", count, data[:count])
```

Index

[Constants](#)

[Variables](#)

[func Chdir\(dir string\) error](#)

[func Chmod\(name string, mode FileMode\) error](#)

[func Chown\(name string, uid, gid int\) error](#)

[func Chtimes\(name string, atime time.Time, mtime time.Time\) error](#)

[func Clearenv\(\)](#)

[func Environ\(\) \[\]string](#)

[func Exit\(code int\)](#)

[func Expand\(s string, mapping func\(string\) string\) string](#)

[func ExpandEnv\(s string\) string](#)

[func Getegid\(\) int](#)

[func Getenv\(key string\) string](#)

[func Geteuid\(\) int](#)

[func Getgid\(\) int](#)

[func Getgroups\(\) \(\[\]int, error\)](#)

[func Getpagesize\(\) int](#)

[func Getpid\(\) int](#)

[func Getppid\(\) int](#)

[func Getuid\(\) int](#)

[func Getwd\(\) \(pwd string, err error\)](#)

[func Hostname\(\) \(name string, err error\)](#)

[func IsExist\(err error\) bool](#)

[func IsNotExist\(err error\) bool](#)

[func IsPathSeparator\(c uint8\) bool](#)

[func IsPermission\(err error\) bool](#)

[func Lchown\(name string, uid, gid int\) error](#)

[func Link\(oldname, newname string\) error](#)

[func Mkdir\(name string, perm FileMode\) error](#)

[func MkdirAll\(path string, perm FileMode\) error](#)

[func NewSyscallError\(syscall string, err error\) error](#)

[func Readlink\(name string\) \(string, error\)](#)

[func Remove\(name string\) error](#)

[func RemoveAll\(path string\) error](#)

[func Rename\(oldname, newname string\) error](#)

[func SameFile\(fi1, fi2 FileInfo\) bool](#)
[func Setenv\(key, value string\) error](#)
[func Symlink\(oldname, newname string\) error](#)
[func TempDir\(\) string](#)
[func Truncate\(name string, size int64\) error](#)
[type File](#)
 [func Create\(name string\) \(file *File, err error\)](#)
 [func NewFile\(fd uintptr, name string\) *File](#)
 [func Open\(name string\) \(file *File, err error\)](#)
 [func OpenFile\(name string, flag int, perm FileMode\) \(file *File, err error\)](#)
 [func Pipe\(\) \(r *File, w *File, err error\)](#)
 [func \(f *File\) Chdir\(\) error](#)
 [func \(f *File\) Chmod\(mode FileMode\) error](#)
 [func \(f *File\) Chown\(uid, gid int\) error](#)
 [func \(f *File\) Close\(\) error](#)
 [func \(f *File\) Fd\(\) uintptr](#)
 [func \(f *File\) Name\(\) string](#)
 [func \(f *File\) Read\(b \[\]byte\) \(n int, err error\)](#)
 [func \(f *File\) ReadAt\(b \[\]byte, off int64\) \(n int, err error\)](#)
 [func \(f *File\) Readdir\(n int\) \(fi \[\]FileInfo, err error\)](#)
 [func \(f *File\) Readdirnames\(n int\) \(names \[\]string, err error\)](#)
 [func \(f *File\) Seek\(offset int64, whence int\) \(ret int64, err error\)](#)
 [func \(f *File\) Stat\(\) \(fi FileInfo, err error\)](#)
 [func \(f *File\) Sync\(\) \(err error\)](#)
 [func \(f *File\) Truncate\(size int64\) error](#)
 [func \(f *File\) Write\(b \[\]byte\) \(n int, err error\)](#)
 [func \(f *File\) WriteAt\(b \[\]byte, off int64\) \(n int, err error\)](#)
 [func \(f *File\) WriteString\(s string\) \(ret int, err error\)](#)
[type FileInfo](#)
 [func Lstat\(name string\) \(fi FileInfo, err error\)](#)
 [func Stat\(name string\) \(fi FileInfo, err error\)](#)
[type FileMode](#)
 [func \(m FileMode\) IsDir\(\) bool](#)
 [func \(m FileMode\) Perm\(\) FileMode](#)
 [func \(m FileMode\) String\(\) string](#)
[type LinkError](#)
 [func \(e *LinkError\) Error\(\) string](#)
[type PathError](#)

[func \(e *PathError\) Error\(\) string](#)
[type ProcAttr](#)
[type Process](#)
[func FindProcess\(pid int\) \(p *Process, err error\)](#)
[func StartProcess\(name string, argv \[\]string, attr *ProcAttr\) \(*Process, error\)](#)
[func \(p *Process\) Kill\(\) error](#)
[func \(p *Process\) Release\(\) error](#)
[func \(p *Process\) Signal\(sig Signal\) error](#)
[func \(p *Process\) Wait\(\) \(*ProcessState, error\)](#)
[type ProcessState](#)
[func \(p *ProcessState\) Exited\(\) bool](#)
[func \(p *ProcessState\) Pid\(\) int](#)
[func \(p *ProcessState\) String\(\) string](#)
[func \(p *ProcessState\) Success\(\) bool](#)
[func \(p *ProcessState\) Sys\(\) interface{}](#)
[func \(p *ProcessState\) SysUsage\(\) interface{}](#)
[func \(p *ProcessState\) SystemTime\(\) time.Duration](#)
[func \(p *ProcessState\) UserTime\(\) time.Duration](#)
[type Signal](#)
[type SyscallError](#)
[func \(e *SyscallError\) Error\(\) string](#)

Package files

[dir_unix.go](#) [doc.go](#) [env.go](#) [error.go](#) [error_posix.go](#) [exec.go](#) [exec_posix.go](#) [exec_unix.go](#) [file.go](#) [file_posix.go](#) [file_unix.go](#) [getwd.go](#) [path.go](#) [path_unix.go](#) [proc.go](#) [stat_linux.go](#) [sys_linux.go](#) [types.go](#)

Constants

```
const (  
    O_RDONLY int = syscall.O_RDONLY // open the file read-only.  
    O_WRONLY int = syscall.O_WRONLY // open the file write-only.  
    O_RDWR  int = syscall.O_RDWR  // open the file read-write.  
    O_APPEND int = syscall.O_APPEND // append data to the file when  
    O_CREATE int = syscall.O_CREAT  // create a new file if none exist  
    O_EXCL   int = syscall.O_EXCL   // used with O_CREATE, file must  
    O_SYNC   int = syscall.O_SYNC   // open for synchronous I/O.  
    O_TRUNC  int = syscall.O_TRUNC  // if possible, truncate file when  
)
```

Flags to Open wrapping those of the underlying system. Not all flags may be implemented on a given system.

```
const (  
    SEEK_SET int = 0 // seek relative to the origin of the file  
    SEEK_CUR int = 1 // seek relative to the current offset  
    SEEK_END int = 2 // seek relative to the end  
)
```

Seek whence values.

```
const (  
    PathSeparator      = '/' // OS-specific path separator  
    PathListSeparator = ':' // OS-specific path list separator  
)  
  
const DevNull = "/dev/null"
```

DevNull is the name of the operating system's “null device.” On Unix-like systems, it is “/dev/null”; on Windows, “NUL”.

Variables

```
var (  
    ErrInvalid      = errors.New("invalid argument")  
    ErrPermission  = errors.New("permission denied")  
    ErrExist       = errors.New("file already exists")  
    ErrNotExist    = errors.New("file does not exist")  
)
```

Portable analogs of some common system call errors.

```
var (  
    Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")  
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")  
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")  
)
```

Stdin, Stdout, and Stderr are open Files pointing to the standard input, standard output, and standard error file descriptors.

```
var Args []string
```

Args hold the command-line arguments, starting with the program name.

func [Chdir](#)

```
func Chdir(dir string) error
```

Chdir changes the current working directory to the named directory. If there is an error, it will be of type `*PathError`.

func [Chmod](#)

```
func Chmod(name string, mode FileMode) error
```

Chmod changes the mode of the named file to mode. If the file is a symbolic link, it changes the mode of the link's target. If there is an error, it will be of type *PathError.

func [Chown](#)

```
func Chown(name string, uid, gid int) error
```

Chown changes the numeric uid and gid of the named file. If the file is a symbolic link, it changes the uid and gid of the link's target. If there is an error, it will be of type `*PathError`.

func [Chtimes](#)

```
func Chtimes(name string, atime time.Time, mtime time.Time) error
```

Chtimes changes the access and modification times of the named file, similar to the Unix `utime()` or `utimes()` functions.

The underlying filesystem may truncate or round the values to a less precise time unit. If there is an error, it will be of type `*PathError`.

func [Clearenv](#)

```
func Clearenv()
```

Clearenv deletes all environment variables.

func Environ

```
func Environ() []string
```

Environ returns a copy of strings representing the environment, in the form "key=value".

func [Exit](#)

```
func Exit(code int)
```

Exit causes the current program to exit with the given status code. Conventionally, code zero indicates success, non-zero an error.

func [Expand](#)

```
func Expand(s string, mapping func(string) string) string
```

Expand replaces `${var}` or `$var` in the string based on the mapping function. Invocations of undefined variables are replaced with the empty string.

func [ExpandEnv](#)

```
func ExpandEnv(s string) string
```

ExpandEnv replaces `${var}` or `$var` in the string according to the values of the current environment variables. References to undefined variables are replaced by the empty string.

func [Getegid](#)

```
func Getegid() int
```

Getegid returns the numeric effective group id of the caller.

func [Getenv](#)

```
func Getenv(key string) string
```

Getenv retrieves the value of the environment variable named by the key. It returns the value, which will be empty if the variable is not present.

func [Geteuid](#)

```
func Geteuid() int
```

Geteuid returns the numeric effective user id of the caller.

func [Getgid](#)

```
func Getgid() int
```

Getgid returns the numeric group id of the caller.

func Getgroups

```
func Getgroups() ([]int, error)
```

Getgroups returns a list of the numeric ids of groups that the caller belongs to.

func [Getpagesize](#)

```
func Getpagesize() int
```

Getpagesize returns the underlying system's memory page size.

func [Getpid](#)

```
func Getpid() int
```

Getpid returns the process id of the caller.

func [Getppid](#)

```
func Getppid() int
```

Getppid returns the process id of the caller's parent.

func [Getuid](#)

```
func Getuid() int
```

Getuid returns the numeric user id of the caller.

func [Getwd](#)

```
func Getwd() (pwd string, err error)
```

Getwd returns a rooted path name corresponding to the current directory. If the current directory can be reached via multiple paths (due to symbolic links), Getwd may return any one of them.

func [Hostname](#)

```
func Hostname() (name string, err error)
```

Hostname returns the host name reported by the kernel.

func [IsExist](#)

```
func IsExist(err error) bool
```

IsExist returns whether the error is known to report that a file or directory already exists. It is satisfied by ErrExist as well as some syscall errors.

func [IsNotExist](#)

```
func IsNotExist(err error) bool
```

IsNotExist returns whether the error is known to report that a file or directory does not exist. It is satisfied by ErrNotExist as well as some syscall errors.

func IsPathSeparator

```
func IsPathSeparator(c uint8) bool
```

IsPathSeparator returns true if c is a directory separator character.

func IsPermission

```
func IsPermission(err error) bool
```

IsPermission returns whether the error is known to report that permission is denied. It is satisfied by ErrPermission as well as some syscall errors.

func [Lchown](#)

```
func Lchown(name string, uid, gid int) error
```

Lchown changes the numeric uid and gid of the named file. If the file is a symbolic link, it changes the uid and gid of the link itself. If there is an error, it will be of type `*PathError`.

func [Link](#)

```
func Link(oldname, newname string) error
```

Link creates newname as a hard link to the oldname file. If there is an error, it will be of type *LinkError.

func [Mkdir](#)

```
func Mkdir(name string, perm FileMode) error
```

Mkdir creates a new directory with the specified name and permission bits. If there is an error, it will be of type *PathError.

func [MkdirAll](#)

```
func MkdirAll(path string, perm FileMode) error
```

MkdirAll creates a directory named path, along with any necessary parents, and returns nil, or else returns an error. The permission bits perm are used for all directories that MkdirAll creates. If path is already a directory, MkdirAll does nothing and returns nil.

func [NewSyscallError](#)

```
func NewSyscallError(syscall string, err error) error
```

NewSyscallError returns, as an error, a new SyscallError with the given system call name and error details. As a convenience, if err is nil, NewSyscallError returns nil.

func [Readlink](#)

```
func Readlink(name string) (string, error)
```

Readlink returns the destination of the named symbolic link. If there is an error, it will be of type `*PathError`.

func [Remove](#)

```
func Remove(name string) error
```

Remove removes the named file or directory. If there is an error, it will be of type `*PathError`.

func [RemoveAll](#)

```
func RemoveAll(path string) error
```

RemoveAll removes path and any children it contains. It removes everything it can but returns the first error it encounters. If the path does not exist, RemoveAll returns nil (no error).

func [Rename](#)

```
func Rename(oldname, newname string) error
```

Rename renames a file.

func [SameFile](#)

```
func SameFile(fi1, fi2 FileInfo) bool
```

SameFile reports whether fi1 and fi2 describe the same file. For example, on Unix this means that the device and inode fields of the two underlying structures are identical; on other systems the decision may be based on the path names. SameFile only applies to results returned by this package's Stat. It returns false in other cases.

func [Setenv](#)

```
func Setenv(key, value string) error
```

Setenv sets the value of the environment variable named by the key. It returns an error, if any.

func [SymLink](#)

```
func SymLink(oldname, newname string) error
```

SymLink creates newname as a symbolic link to oldname. If there is an error, it will be of type *LinkError.

func [TempDir](#)

```
func TempDir() string
```

TempDir returns the default directory to use for temporary files.

func [Truncate](#)

```
func Truncate(name string, size int64) error
```

Truncate changes the size of the named file. If the file is a symbolic link, it changes the size of the link's target. If there is an error, it will be of type `*PathError`.

type [File](#)

```
type File struct {  
    // contains filtered or unexported fields  
}
```

File represents an open file descriptor.

func [Create](#)

```
func Create(name string) (file *File, err error)
```

Create creates the named file mode 0666 (before umask), truncating it if it already exists. If successful, methods on the returned File can be used for I/O; the associated file descriptor has mode O_RDWR. If there is an error, it will be of type *PathError.

func [NewFile](#)

```
func NewFile(fd uintptr, name string) *File
```

NewFile returns a new File with the given file descriptor and name.

func [Open](#)

```
func Open(name string) (file *File, err error)
```

Open opens the named file for reading. If successful, methods on the returned file can be used for reading; the associated file descriptor has mode O_RDONLY. If there is an error, it will be of type *PathError.

func [OpenFile](#)

```
func OpenFile(name string, flag int, perm FileMode) (file *File, err
```

OpenFile is the generalized open call; most users will use Open or Create instead. It opens the named file with specified flag (O_RDONLY etc.) and perm, (0666 etc.) if applicable. If successful, methods on the returned File can be used for I/O. If there is an error, it will be of type *PathError.

func [Pipe](#)

```
func Pipe() (r *File, w *File, err error)
```

Pipe returns a connected pair of Files; reads from r return bytes written to w. It returns the files and an error, if any.

func (*File) [Chdir](#)

```
func (f *File) Chdir() error
```

Chdir changes the current working directory to the file, which must be a directory. If there is an error, it will be of type *PathError.

func (*File) [Chmod](#)

```
func (f *File) Chmod(mode FileMode) error
```

Chmod changes the mode of the file to mode. If there is an error, it will be of type *PathError.

func (*File) [Chown](#)

```
func (f *File) Chown(uid, gid int) error
```

Chown changes the numeric uid and gid of the named file. If there is an error, it will be of type *PathError.

func (*File) [Close](#)

```
func (f *File) Close() error
```

Close closes the File, rendering it unusable for I/O. It returns an error, if any.

func (*File) [Fd](#)

```
func (f *File) Fd() uintptr
```

Fd returns the integer Unix file descriptor referencing the open file.

func (*File) [Name](#)

```
func (f *File) Name() string
```

Name returns the name of the file as presented to Open.

func (*File) [Read](#)

```
func (f *File) Read(b []byte) (n int, err error)
```

Read reads up to len(b) bytes from the File. It returns the number of bytes read and an error, if any. EOF is signaled by a zero count with err set to io.EOF.

func (*File) [ReadAt](#)

```
func (f *File) ReadAt(b []byte, off int64) (n int, err error)
```

ReadAt reads len(b) bytes from the File starting at byte offset off. It returns the number of bytes read and the error, if any. ReadAt always returns a non-nil error when $n < \text{len}(b)$. At end of file, that error is io.EOF.

func (*File) [Readdir](#)

```
func (f *File) Readdir(n int) (fi []FileInfo, err error)
```

Readdir reads the contents of the directory associated with file and returns an array of up to n FileInfo values, as would be returned by Lstat, in directory order. Subsequent calls on the same file will yield further FileInfos.

If $n > 0$, Readdir returns at most n FileInfo structures. In this case, if Readdir returns an empty slice, it will return a non-nil error explaining why. At the end of a directory, the error is io.EOF.

If $n \leq 0$, Readdir returns all the FileInfo from the directory in a single slice. In this case, if Readdir succeeds (reads all the way to the end of the directory), it returns the slice and a nil error. If it encounters an error before the end of the directory, Readdir returns the FileInfo read until that point and a non-nil error.

func (*File) [Readdirnames](#)

```
func (f *File) Readdirnames(n int) (names []string, err error)
```

Readdirnames reads and returns a slice of names from the directory f.

If $n > 0$, Readdirnames returns at most n names. In this case, if Readdirnames returns an empty slice, it will return a non-nil error explaining why. At the end of a directory, the error is `io.EOF`.

If $n \leq 0$, Readdirnames returns all the names from the directory in a single slice. In this case, if Readdirnames succeeds (reads all the way to the end of the directory), it returns the slice and a nil error. If it encounters an error before the end of the directory, Readdirnames returns the names read until that point and a non-nil error.

func (*File) [Seek](#)

```
func (f *File) Seek(offset int64, whence int) (ret int64, err error)
```

Seek sets the offset for the next Read or Write on file to offset, interpreted according to whence: 0 means relative to the origin of the file, 1 means relative to the current offset, and 2 means relative to the end. It returns the new offset and an error, if any.

func (*File) [Stat](#)

```
func (f *File) Stat() (fi FileInfo, err error)
```

Stat returns the FileInfo structure describing file. If there is an error, it will be of type `*PathError`.

func (*File) [Sync](#)

```
func (f *File) Sync() (err error)
```

Sync commits the current contents of the file to stable storage. Typically, this means flushing the file system's in-memory copy of recently written data to disk.

func (*File) [Truncate](#)

```
func (f *File) Truncate(size int64) error
```

Truncate changes the size of the file. It does not change the I/O offset. If there is an error, it will be of type `*PathError`.

func (*File) [Write](#)

```
func (f *File) Write(b []byte) (n int, err error)
```

Write writes `len(b)` bytes to the File. It returns the number of bytes written and an error, if any. Write returns a non-nil error when `n != len(b)`.

func (*File) [WriteAt](#)

```
func (f *File) WriteAt(b []byte, off int64) (n int, err error)
```

WriteAt writes `len(b)` bytes to the File starting at byte offset `off`. It returns the number of bytes written and an error, if any. WriteAt returns a non-nil error when `n != len(b)`.

func (*File) [WriteString](#)

```
func (f *File) WriteString(s string) (ret int, err error)
```

WriteString is like Write, but writes the contents of string `s` rather than an array of bytes.

type [FileInfo](#)

```
type FileInfo interface {  
    Name() string        // base name of the file  
    Size() int64         // length in bytes for regular files; system  
    Mode() FileMode     // file mode bits  
    ModTime() time.Time // modification time  
    IsDir() bool        // abbreviation for Mode().IsDir()  
    Sys() interface{}   // underlying data source (can return nil)  
}
```

A FileInfo describes a file and is returned by Stat and Lstat

func [Lstat](#)

```
func Lstat(name string) (fi FileInfo, err error)
```

Lstat returns a FileInfo describing the named file. If the file is a symbolic link, the returned FileInfo describes the symbolic link. Lstat makes no attempt to follow the link. If there is an error, it will be of type *PathError.

func [Stat](#)

```
func Stat(name string) (fi FileInfo, err error)
```

Stat returns a FileInfo describing the named file. If there is an error, it will be of type *PathError.

type [FileMode](#)

```
type FileMode uint32
```

A FileMode represents a file's mode and permission bits. The bits have the same definition on all systems, so that information about files can be moved from one system to another portably. Not all bits apply to all systems. The only required bit is ModeDir for directories.

```
const (  
    // The single letters are the abbreviations  
    // used by the String method's formatting.  
    ModeDir          FileMode = 1 << (32 - 1 - iota) // d: is a direct  
    ModeAppend       // a: append-only  
    ModeExclusive    // l: exclusive u  
    ModeTemporary    // T: temporary f  
    ModeSymlink      // L: symbolic li  
    ModeDevice       // D: device file  
    ModeNamedPipe    // p: named pipe  
    ModeSocket       // S: Unix domain  
    ModeSetuid       // u: setuid  
    ModeSetgid       // g: setgid  
    ModeCharDevice   // c: Unix charac  
    ModeSticky  
  
    // Mask for the type bits. For regular files, none will be set.  
    ModeType = ModeDir | ModeSymlink | ModeNamedPipe | ModeSocket |  
  
    ModePerm FileMode = 0777 // permission bits  
)
```

The defined file mode bits are the most significant bits of the FileMode. The nine least-significant bits are the standard Unix `rw-rw-rw-` permissions. The values of these bits should be considered part of the public API and may be used in wire protocols or disk representations: they must not be changed, although new bits might be added.

func (FileMode) [IsDir](#)

```
func (m FileMode) IsDir() bool
```

IsDir reports whether `m` describes a directory. That is, it tests for the ModeDir bit being set in `m`.

func (FileMode) [Perm](#)

```
func (m FileMode) Perm() FileMode
```

Perm returns the Unix permission bits in m.

func (FileMode) [String](#)

```
func (m FileMode) String() string
```

type [LinkError](#)

```
type LinkError struct {  
    Op string  
    Old string  
    New string  
    Err error  
}
```

LinkError records an error during a link or symlink or rename system call and the paths that caused it.

func (*LinkError) [Error](#)

```
func (e *LinkError) Error() string
```

type [PathError](#)

```
type PathError struct {  
    Op    string  
    Path string  
    Err  error  
}
```

PathError records an error and the operation and file path that caused it.

func (*PathError) [Error](#)

```
func (e *PathError) Error() string
```

type [ProcAttr](#)

```
type ProcAttr struct {
    // If Dir is non-empty, the child changes into the directory before
    // creating the process.
    Dir string
    // If Env is non-nil, it gives the environment variables for the
    // new process in the form returned by Environ.
    // If it is nil, the result of Environ will be used.
    Env []string
    // Files specifies the open files inherited by the new process.
    // first three entries correspond to standard input, standard output,
    // standard error. An implementation may support additional entries
    // depending on the underlying operating system. A nil entry corresponds
    // to that file being closed when the process starts.
    Files []*File

    // Operating system-specific process creation attributes.
    // Note that setting this field means that your program
    // may not execute properly or even compile on some
    // operating systems.
    Sys *syscall.SysProcAttr
}
```

ProcAttr holds the attributes that will be applied to a new process started by StartProcess.

type [Process](#)

```
type Process struct {
    Pid int
    // contains filtered or unexported fields
}
```

Process stores the information about a process created by StartProcess.

func [FindProcess](#)

```
func FindProcess(pid int) (p *Process, err error)
```

FindProcess looks for a running process by its pid. The Process it returns can be used to obtain information about the underlying operating system process.

func [StartProcess](#)

```
func StartProcess(name string, argv []string, attr *ProcAttr) (*Proc
```

StartProcess starts a new process with the program, arguments and attributes specified by name, argv and attr.

StartProcess is a low-level interface. The os/exec package provides higher-level interfaces.

If there is an error, it will be of type *PathError.

func (*Process) [Kill](#)

```
func (p *Process) Kill() error
```

Kill causes the Process to exit immediately.

func (*Process) [Release](#)

```
func (p *Process) Release() error
```

Release releases any resources associated with the Process p, rendering it

unusable in the future. Release only needs to be called if Wait is not.

func (*Process) [Signal](#)

```
func (p *Process) Signal(sig Signal) error
```

Signal sends a signal to the Process.

func (*Process) [Wait](#)

```
func (p *Process) Wait() (*ProcessState, error)
```

Wait waits for the Process to exit, and then returns a ProcessState describing its status and an error, if any. Wait releases any resources associated with the Process.

type [ProcessState](#)

```
type ProcessState struct {  
    // contains filtered or unexported fields  
}
```

ProcessState stores information about a process, as reported by Wait.

func (***ProcessState**) [Exited](#)

```
func (p *ProcessState) Exited() bool
```

Exited returns whether the program has exited.

func (***ProcessState**) [Pid](#)

```
func (p *ProcessState) Pid() int
```

Pid returns the process id of the exited process.

func (***ProcessState**) [String](#)

```
func (p *ProcessState) String() string
```

func (***ProcessState**) [Success](#)

```
func (p *ProcessState) Success() bool
```

Success reports whether the program exited successfully, such as with exit status 0 on Unix.

func (***ProcessState**) [Sys](#)

```
func (p *ProcessState) Sys() interface{}
```

Sys returns system-dependent exit information about the process. Convert it to the appropriate underlying type, such as syscall.WaitStatus on Unix, to access its contents.

func (*ProcessState) [SysUsage](#)

```
func (p *ProcessState) SysUsage() interface{}
```

SysUsage returns system-dependent resource usage information about the exited process. Convert it to the appropriate underlying type, such as `*syscall.Rusage` on Unix, to access its contents.

func (*ProcessState) [SystemTime](#)

```
func (p *ProcessState) SystemTime() time.Duration
```

SystemTime returns the system CPU time of the exited process and its children.

func (*ProcessState) [UserTime](#)

```
func (p *ProcessState) UserTime() time.Duration
```

UserTime returns the user CPU time of the exited process and its children.

type [Signal](#)

```
type Signal interface {
    String() string
    Signal() // to distinguish from other Stringers
}
```

A Signal represents an operating system signal. The usual underlying implementation is operating system-dependent: on Unix it is syscall.Signal.

```
var (
    Interrupt Signal = syscall.SIGINT
    Kill       Signal = syscall.SIGKILL
)
```

The only signal values guaranteed to be present on all systems are Interrupt (send the process an interrupt) and Kill (force the process to exit).

type [SyscallError](#)

```
type SyscallError struct {  
    Syscall string  
    Err     error  
}
```

SyscallError records an error from a specific system call.

func (*SyscallError) [Error](#)

```
func (e *SyscallError) Error() string
```

Subdirectories

Name	Synopsis
------	----------

exec	Package exec runs external commands.
----------------------	--------------------------------------

signal	Package signal implements access to incoming signals.
------------------------	---

user	Package user allows user account lookups by name or id.
----------------------	---

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package exec

```
import "os/exec"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package exec runs external commands. It wraps `os.StartProcess` to make it easier to remap stdin and stdout, connect I/O with pipes, and do other adjustments.

Index

Variables

[func LookPath\(file string\) \(string, error\)](#)

type Cmd

[func Command\(name string, arg ...string\) *Cmd](#)

[func \(c *Cmd\) CombinedOutput\(\) \(\[\]byte, error\)](#)

[func \(c *Cmd\) Output\(\) \(\[\]byte, error\)](#)

[func \(c *Cmd\) Run\(\) error](#)

[func \(c *Cmd\) Start\(\) error](#)

[func \(c *Cmd\) StderrPipe\(\) \(io.ReadCloser, error\)](#)

[func \(c *Cmd\) StdinPipe\(\) \(io.WriteCloser, error\)](#)

[func \(c *Cmd\) StdoutPipe\(\) \(io.ReadCloser, error\)](#)

[func \(c *Cmd\) Wait\(\) error](#)

type Error

[func \(e *Error\) Error\(\) string](#)

type ExitError

[func \(e *ExitError\) Error\(\) string](#)

Examples

[Cmd.Output](#)

[Cmd.Start](#)

[Cmd.StdoutPipe](#)

[Command](#)

[LookPath](#)

Package files

[exec.go](#) [lp_unix.go](#)

Variables

```
var ErrNotFound = errors.New("executable file not found in $PATH")
```

ErrNotFound is the error resulting if a path search failed to find an executable file.

func [LookPath](#)

```
func LookPath(file string) (string, error)
```

LookPath searches for an executable binary named file in the directories named by the PATH environment variable. If file contains a slash, it is tried directly and the PATH is not consulted.

? Example

? Example

Code:

```
path, err := exec.LookPath("fortune")
if err != nil {
    log.Fatal("installing fortune is in your future")
}
fmt.Printf("fortune is available at %s\n", path)
```

type Cmd

```
type Cmd struct {
    // Path is the path of the command to run.
    //
    // This is the only field that must be set to a non-zero
    // value.
    Path string

    // Args holds command line arguments, including the command as A
    // If the Args field is empty or nil, Run uses {Path}.
    //
    // In typical use, both Path and Args are set by calling Command
    Args []string

    // Env specifies the environment of the process.
    // If Env is nil, Run uses the current process's environment.
    Env []string

    // Dir specifies the working directory of the command.
    // If Dir is the empty string, Run runs the command in the
    // calling process's current directory.
    Dir string

    // Stdin specifies the process's standard input. If Stdin is
    // nil, the process reads from the null device (os.DevNull).
    Stdin io.Reader

    // Stdout and Stderr specify the process's standard output and e
    //
    // If either is nil, Run connects the corresponding file descrip
    // to the null device (os.DevNull).
    //
    // If Stdout and Stderr are the same writer, at most one
    // goroutine at a time will call Write.
    Stdout io.Writer
    Stderr io.Writer

    // ExtraFiles specifies additional open files to be inherited by
    // new process. It does not include standard input, standard out
    // standard error. If non-nil, entry i becomes file descriptor 3
    //
    // BUG: on OS X 10.6, child processes may sometimes inherit unwa
    // http://golang.org/issue/2603
    ExtraFiles []*os.File

    // SysProcAttr holds optional, operating system-specific attribu
```

```

// Run passes it to os.StartProcess as the os.ProcAttr's Sys file
SysProcAttr *syscall.SysProcAttr

// Process is the underlying process, once started.
Process *os.Process

// ProcessState contains information about an exited process,
// available after a call to Wait or Run.
ProcessState *os.ProcessState
// contains filtered or unexported fields
}

```

Cmd represents an external command being prepared or run.

func [Command](#)

```
func Command(name string, arg ...string) *Cmd
```

Command returns the Cmd struct to execute the named program with the given arguments.

It sets Path and Args in the returned structure and zeroes the other fields.

If name contains no path separators, Command uses LookPath to resolve the path to a complete name if possible. Otherwise it uses name directly.

The returned Cmd's Args field is constructed from the command name followed by the elements of arg, so arg should not include the command name itself. For example, Command("echo", "hello")

? Example

? Example

Code:

```

cmd := exec.Command("tr", "a-z", "A-Z")
cmd.Stdin = strings.NewReader("some input")
var out bytes.Buffer
cmd.Stdout = &out
err := cmd.Run()
if err != nil {
    log.Fatal(err)
}

```

```
fmt.Printf("in all caps: %q\n", out.String())
```

func (*Cmd) [CombinedOutput](#)

```
func (c *Cmd) CombinedOutput() ([]byte, error)
```

CombinedOutput runs the command and returns its combined standard output and standard error.

func (*Cmd) [Output](#)

```
func (c *Cmd) Output() ([]byte, error)
```

Output runs the command and returns its standard output.

? Example

? Example

Code:

```
out, err := exec.Command("date").Output()
if err != nil {
    log.Fatal(err)
}
fmt.Printf("The date is %s\n", out)
```

func (*Cmd) [Run](#)

```
func (c *Cmd) Run() error
```

Run starts the specified command and waits for it to complete.

The returned error is nil if the command runs, has no problems copying stdin, stdout, and stderr, and exits with a zero exit status.

If the command fails to run or doesn't complete successfully, the error is of type *ExitError. Other error types may be returned for I/O problems.

func (*Cmd) [Start](#)

```
func (c *Cmd) Start() error
```

Start starts the specified command but does not wait for it to complete.

? Example

? Example

Code:

```
cmd := exec.Command("sleep", "5")
err := cmd.Start()
if err != nil {
    log.Fatal(err)
}
log.Printf("Waiting for command to finish...")
err = cmd.Wait()
log.Printf("Command finished with error: %v", err)
```

func (*Cmd) [StderrPipe](#)

```
func (c *Cmd) StderrPipe() (io.ReadCloser, error)
```

StderrPipe returns a pipe that will be connected to the command's standard error when the command starts. The pipe will be closed automatically after Wait sees the command exit.

func (*Cmd) [StdinPipe](#)

```
func (c *Cmd) StdinPipe() (io.WriteCloser, error)
```

StdinPipe returns a pipe that will be connected to the command's standard input when the command starts.

func (*Cmd) [StdoutPipe](#)

```
func (c *Cmd) StdoutPipe() (io.ReadCloser, error)
```

StdoutPipe returns a pipe that will be connected to the command's standard output when the command starts. The pipe will be closed automatically after Wait sees the command exit.

? Example

? Example

Code:

```
cmd := exec.Command("echo", "-n", `{"Name": "Bob", "Age": 32}`)
stdout, err := cmd.StdoutPipe()
if err != nil {
    log.Fatal(err)
}
if err := cmd.Start(); err != nil {
    log.Fatal(err)
}
var person struct {
    Name string
    Age  int
}
if err := json.NewDecoder(stdout).Decode(&person); err != nil {
    log.Fatal(err)
}
if err := cmd.Wait(); err != nil {
    log.Fatal(err)
}
fmt.Printf("%s is %d years old\n", person.Name, person.Age)
```

func (*Cmd) [Wait](#)

```
func (c *Cmd) Wait() error
```

Wait waits for the command to exit. It must have been started by Start.

The returned error is nil if the command runs, has no problems copying stdin, stdout, and stderr, and exits with a zero exit status.

If the command fails to run or doesn't complete successfully, the error is of type *ExitError. Other error types may be returned for I/O problems.

type [Error](#)

```
type Error struct {  
    Name string  
    Err  error  
}
```

Error records the name of a binary that failed to be executed and the reason it failed.

func (*Error) [Error](#)

```
func (e *Error) Error() string
```

type [ExitError](#)

```
type ExitError struct {  
    *os.ProcessState  
}
```

An ExitError reports an unsuccessful exit by a command.

func (***ExitError**) [Error](#)

```
func (e *ExitError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package signal

```
import "os/signal"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package signal implements access to incoming signals.

Index

[func Notify\(c chan<- os.Signal, sig ...os.Signal\)](#)

[Bugs](#)

Package files

[signal.go](#) [signal_unix.go](#)

func [Notify](#)

```
func Notify(c chan<- os.Signal, sig ...os.Signal)
```

Notify causes package signal to relay incoming signals to c. If no signals are listed, all incoming signals will be relayed to c. Otherwise, just the listed signals will.

Package signal will not block sending to c: the caller must ensure that c has sufficient buffer space to keep up with the expected signal rate. For a channel used for notification of just one signal value, a buffer of size 1 is sufficient.

Bugs

This package is not yet implemented on Plan 9 and Windows.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package user

```
import "os/user"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package user allows user account lookups by name or id.

Index

[type UnknownUserError](#)

[func \(e UnknownUserError\) Error\(\) string](#)

[type UnknownUserIdError](#)

[func \(e UnknownUserIdError\) Error\(\) string](#)

[type User](#)

[func Current\(\) \(*User, error\)](#)

[func Lookup\(username string\) \(*User, error\)](#)

[func LookupId\(uid string\) \(*User, error\)](#)

Package files

[lookup_unix.go](#) [user.go](#)

type UnknownUserError

```
type UnknownUserError string
```

UnknownUserError is returned by Lookup when a user cannot be found.

func (UnknownUserError) Error

```
func (e UnknownUserError) Error() string
```

type [UnknownUserIdError](#)

```
type UnknownUserIdError int
```

UnknownUserIdError is returned by LookupId when a user cannot be found.

func (UnknownUserIdError) [Error](#)

```
func (e UnknownUserIdError) Error() string
```

type [User](#)

```
type User struct {
    Uid      string // user id
    Gid      string // primary group id
    Username string
    Name     string
    HomeDir  string
}
```

User represents a user account.

On posix systems Uid and Gid contain a decimal number representing uid and gid. On windows Uid and Gid contain security identifier (SID) in a string format.

func [Current](#)

```
func Current() (*User, error)
```

Current returns the current user.

func [Lookup](#)

```
func Lookup(username string) (*User, error)
```

Lookup looks up a user by username. If the user cannot be found, the returned error is of type `UnknownUserError`.

func [LookupId](#)

```
func LookupId(uid string) (*User, error)
```

LookupId looks up a user by userid. If the user cannot be found, the returned error is of type `UnknownUserIdError`.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package path

```
import "path"
```

[Overview](#)

[Index](#)

[Examples](#)

[Subdirectories](#)

Overview ?

Overview ?

Package path implements utility routines for manipulating slash-separated paths.

Index

Variables

[func Base\(path string\) string](#)

[func Clean\(path string\) string](#)

[func Dir\(path string\) string](#)

[func Ext\(path string\) string](#)

[func IsAbs\(path string\) bool](#)

[func Join\(elem ...string\) string](#)

[func Match\(pattern, name string\) \(matched bool, err error\)](#)

[func Split\(path string\) \(dir, file string\)](#)

Examples

[Base](#)

[Clean](#)

[Dir](#)

[Ext](#)

[IsAbs](#)

[Join](#)

[Split](#)

Package files

[match.go](#) [path.go](#)

Variables

```
var ErrBadPattern = errors.New("syntax error in pattern")
```

ErrBadPattern indicates a globbing pattern was malformed.

func [Base](#)

```
func Base(path string) string
```

Base returns the last element of path. Trailing slashes are removed before extracting the last element. If the path is empty, Base returns ".". If the path consists entirely of slashes, Base returns "/".

? Example

? Example

Code:

```
fmt.Println(path.Base("/a/b"))
```

Output:

b

func [Clean](#)

```
func Clean(path string) string
```

Clean returns the shortest path name equivalent to path by purely lexical processing. It applies the following rules iteratively until no further processing can be done:

1. Replace multiple slashes with a single slash.
2. Eliminate each . path name element (the current directory).
3. Eliminate each inner .. path name element (the parent directory) along with the non-.. element that precedes it.
4. Eliminate .. elements that begin a rooted path: that is, replace "../" by "/" at the beginning of a path.

The returned path ends in a slash only if it is the root "/".

If the result of this process is an empty string, Clean returns the string ".".

See also Rob Pike, "Lexical File Names in Plan 9 or Getting Dot-Dot Right," <http://plan9.bell-labs.com/sys/doc/lexnames.html>

? Example

? Example

Code:

```
paths := []string{
    "a/c",
    "a//c",
    "a/c/.",
    "a/c/b/..",
    "../a/c",
    "../a/b/../../../../c",
}

for _, p := range paths {
    fmt.Printf("Clean(%q) = %q\n", p, path.Clean(p))
}
```

Output:

```
Clean("a/c") = "a/c"  
Clean("a//c") = "a/c"  
Clean("a/c/.") = "a/c"  
Clean("a/c/b/..") = "a/c"  
Clean("../a/c") = "/a/c"  
Clean("../a/b/../../c") = "/a/c"
```

func [Dir](#)

```
func Dir(path string) string
```

Dir returns all but the last element of path, typically the path's directory. The path is Cleaned and trailing slashes are removed before processing. If the path is empty, Dir returns ".". If the path consists entirely of slashes followed by non-slash bytes, Dir returns a single slash. In any other case, the returned path does not end in a slash.

? Example

? Example

Code:

```
fmt.Println(path.Dir("/a/b/c"))
```

Output:

```
/a/b
```

func [Ext](#)

```
func Ext(path string) string
```

Ext returns the file name extension used by path. The extension is the suffix beginning at the final dot in the final slash-separated element of path; it is empty if there is no dot.

? Example

? Example

Code:

```
fmt.Println(path.Ext("/a/b/c/bar.css"))
```

Output:

```
.css
```

func [IsAbs](#)

```
func IsAbs(path string) bool
```

IsAbs returns true if the path is absolute.

? Example

? Example

Code:

```
fmt.Println(path.IsAbs("/dev/null"))
```

Output:

```
true
```

func [Join](#)

```
func Join(elem ...string) string
```

Join joins any number of path elements into a single path, adding a separating slash if necessary. The result is Cleaned; in particular, all empty strings are ignored.

? Example

? Example

Code:

```
fmt.Println(path.Join("a", "b", "c"))
```

Output:

a/b/c

func Match

func Match(pattern, name string) (matched bool, err error)

Match returns true if name matches the shell file name pattern. The pattern syntax is:

pattern:

{ term }

term:

'*'	matches any sequence of non-/ characters
'?'	matches any single non-/ character
'[' ['^']	{ character-range } ']'
	character class (must be non-empty)
c	matches character c (c != '*', '?', '\\', '[')
'\\' c	matches character c

character-range:

c	matches character c (c != '\\', '-', ']')
'\\' c	matches character c
lo '-' hi	matches character c for lo <= c <= hi

Match requires pattern to match all of name, not just a substring. The only possible returned error is ErrBadPattern, when pattern is malformed.

func [Split](#)

```
func Split(path string) (dir, file string)
```

Split splits path immediately following the final slash. separating it into a directory and file name component. If there is no slash path, Split returns an empty dir and file set to path. The returned values have the property that path = dir+file.

? Example

? Example

Code:

```
fmt.Println(path.Split("static/myfile.css"))
```

Output:

```
static/ myfile.css
```

Subdirectories

Name	Synopsis
------	----------

filepath	Package filepath implements utility routines for manipulating filename paths in a way compatible with the target operating system-defined file paths.
--------------------------	---

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package filepath

```
import "path/filepath"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package filepath implements utility routines for manipulating filename paths in a way compatible with the target operating system-defined file paths.

Index

Constants

Variables

[func Abs\(path string\) \(string, error\)](#)

[func Base\(path string\) string](#)

[func Clean\(path string\) string](#)

[func Dir\(path string\) string](#)

[func EvalSymlinks\(path string\) \(string, error\)](#)

[func Ext\(path string\) string](#)

[func FromSlash\(path string\) string](#)

[func Glob\(pattern string\) \(matches \[\]string, err error\)](#)

[func HasPrefix\(p, prefix string\) bool](#)

[func IsAbs\(path string\) bool](#)

[func Join\(elem ...string\) string](#)

[func Match\(pattern, name string\) \(matched bool, err error\)](#)

[func Rel\(basepath, targpath string\) \(string, error\)](#)

[func Split\(path string\) \(dir, file string\)](#)

[func SplitList\(path string\) \[\]string](#)

[func ToSlash\(path string\) string](#)

[func VolumeName\(path string\) string](#)

[func Walk\(root string, walkFn WalkFunc\) error](#)

[type WalkFunc](#)

Package files

[match.go](#) [path.go](#) [path_unix.go](#) [symlink.go](#)

Constants

```
const (  
    Separator      = os.PathSeparator  
    ListSeparator = os.PathListSeparator  
)
```

Variables

```
var ErrBadPattern = errors.New("syntax error in pattern")
```

ErrBadPattern indicates a globbing pattern was malformed.

```
var SkipDir = errors.New("skip this directory")
```

SkipDir is used as a return value from WalkFuncs to indicate that the directory named in the call is to be skipped. It is not returned as an error by any function.

func [Abs](#)

```
func Abs(path string) (string, error)
```

Abs returns an absolute representation of path. If the path is not absolute it will be joined with the current working directory to turn it into an absolute path. The absolute path name for a given file is not guaranteed to be unique.

func Base

```
func Base(path string) string
```

Base returns the last element of path. Trailing path separators are removed before extracting the last element. If the path is empty, Base returns ".". If the path consists entirely of separators, Base returns a single separator.

func Clean

```
func Clean(path string) string
```

Clean returns the shortest path name equivalent to path by purely lexical processing. It applies the following rules iteratively until no further processing can be done:

1. Replace multiple Separator elements with a single one.
2. Eliminate each . path name element (the current directory)
3. Eliminate each inner .. path name element (the parent dir along with the non-.. element that precedes it.)
4. Eliminate .. elements that begin a rooted path: that is, replace "../.." by "/" at the beginning of a path, assuming Separator is '/'.

The returned path ends in a slash only if it represents a root directory, such as "/" on Unix or `C:\` on Windows.

If the result of this process is an empty string, Clean returns the string ".".

See also Rob Pike, "Lexical File Names in Plan 9 or Getting Dot-Dot Right," <http://plan9.bell-labs.com/sys/doc/lexnames.html>

func [Dir](#)

```
func Dir(path string) string
```

Dir returns all but the last element of path, typically the path's directory. Trailing path separators are removed before processing. If the path is empty, Dir returns ".". If the path consists entirely of separators, Dir returns a single separator. The returned path does not end in a separator unless it is the root directory.

func [EvalSymlinks](#)

```
func EvalSymlinks(path string) (string, error)
```

EvalSymlinks returns the path name after the evaluation of any symbolic links. If path is relative the result will be relative to the current directory, unless one of the components is an absolute symbolic link.

func [Ext](#)

```
func Ext(path string) string
```

Ext returns the file name extension used by path. The extension is the suffix beginning at the final dot in the final element of path; it is empty if there is no dot.

func [FromSlash](#)

```
func FromSlash(path string) string
```

FromSlash returns the result of replacing each slash (/) character in path with a separator character. Multiple slashes are replaced by multiple separators.

func [Glob](#)

```
func Glob(pattern string) (matches []string, err error)
```

Glob returns the names of all files matching pattern or nil if there is no matching file. The syntax of patterns is the same as in Match. The pattern may describe hierarchical names such as /usr/*/bin/ed (assuming the Separator is '/').

func [HasPrefix](#)

```
func HasPrefix(p, prefix string) bool
```

HasPrefix exists for historical compatibility and should not be used.

func [IsAbs](#)

```
func IsAbs(path string) bool
```

IsAbs returns true if the path is absolute.

func [Join](#)

```
func Join(elem ...string) string
```

Join joins any number of path elements into a single path, adding a Separator if necessary. The result is Cleaned, in particular all empty strings are ignored.

func [Match](#)

func Match(pattern, name string) (matched bool, err error)

Match returns true if name matches the shell file name pattern. The pattern syntax is:

pattern:

{ term }

term:

'*'	matches any sequence of non-Separator characters
'?'	matches any single non-Separator character
'[' ['^']	{ character-range } ']'
	character class (must be non-empty)
c	matches character c (c != '*', '?', '\\', '[')
'\\' c	matches character c

character-range:

c	matches character c (c != '\\', '-', ']')
'\\' c	matches character c
lo '-' hi	matches character c for lo <= c <= hi

Match requires pattern to match all of name, not just a substring. The only possible returned error is ErrBadPattern, when pattern is malformed.

On Windows, escaping is disabled. Instead, '\\' is treated as path separator.

func [Rel](#)

```
func Rel(basepath, targpath string) (string, error)
```

Rel returns a relative path that is lexically equivalent to targpath when joined to basepath with an intervening separator. That is, `Join(basepath, Rel(basepath, targpath))` is equivalent to targpath itself. On success, the returned path will always be relative to basepath, even if basepath and targpath share no elements. An error is returned if targpath can't be made relative to basepath or if knowing the current working directory would be necessary to compute it.

func [Split](#)

```
func Split(path string) (dir, file string)
```

Split splits path immediately following the final Separator, separating it into a directory and file name component. If there is no Separator in path, Split returns an empty dir and file set to path. The returned values have the property that path = dir+file.

func [SplitList](#)

```
func SplitList(path string) []string
```

SplitList splits a list of paths joined by the OS-specific ListSeparator, usually found in PATH or GOPATH environment variables. Unlike strings.Split, SplitList returns an empty slice when passed an empty string.

func [ToSlash](#)

```
func ToSlash(path string) string
```

ToSlash returns the result of replacing each separator character in path with a slash ('/') character. Multiple separators are replaced by multiple slashes.

func [VolumeName](#)

```
func VolumeName(path string) string
```

VolumeName returns the leading volume name on Windows. It returns "" elsewhere.

func [Walk](#)

```
func Walk(root string, walkFn WalkFunc) error
```

Walk walks the file tree rooted at root, calling walkFn for each file or directory in the tree, including root. All errors that arise visiting files and directories are filtered by walkFn. The files are walked in lexical order, which makes the output deterministic but means that for very large directories Walk can be inefficient.

type [WalkFunc](#)

```
type WalkFunc func(path string, info os.FileInfo, err error) error
```

WalkFunc is the type of the function called for each file or directory visited by Walk. If there was a problem walking to the file or directory named by path, the incoming error will describe the problem and the function can decide how to handle that error (and Walk will not descend into that directory). If an error is returned, processing stops. The sole exception is that if path is a directory and the function returns the special value SkipDir, the contents of the directory are skipped and processing continues as usual on the next file.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package reflect

```
import "reflect"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `reflect` implements run-time reflection, allowing a program to manipulate objects with arbitrary types. The typical use is to take a value with static type `interface{}` and extract its dynamic type information by calling `TypeOf`, which returns a `Type`.

A call to `ValueOf` returns a `Value` representing the run-time data. `Zero` takes a `Type` and returns a `Value` representing a zero value for that type.

See "The Laws of Reflection" for an introduction to reflection in Go:
http://golang.org/doc/articles/laws_of_reflection.html

Index

[func Copy\(dst, src Value\) int](#)
[func DeepEqual\(a1, a2 interface{}\) bool](#)
[type ChanDir](#)
 [func \(d ChanDir\) String\(\) string](#)
[type Kind](#)
 [func \(k Kind\) String\(\) string](#)
[type Method](#)
[type SliceHeader](#)
[type StringHeader](#)
[type StructField](#)
[type StructTag](#)
 [func \(tag StructTag\) Get\(key string\) string](#)
[type Type](#)
 [func PtrTo\(t Type\) Type](#)
 [func TypeOf\(i interface{}\) Type](#)
[type Value](#)
 [func Append\(s Value, x ...Value\) Value](#)
 [func AppendSlice\(s, t Value\) Value](#)
 [func Indirect\(v Value\) Value](#)
 [func MakeChan\(typ Type, buffer int\) Value](#)
 [func MakeMap\(typ Type\) Value](#)
 [func MakeSlice\(typ Type, len, cap int\) Value](#)
 [func New\(typ Type\) Value](#)
 [func NewAt\(typ Type, p unsafe.Pointer\) Value](#)
 [func ValueOf\(i interface{}\) Value](#)
 [func Zero\(typ Type\) Value](#)
 [func \(v Value\) Addr\(\) Value](#)
 [func \(v Value\) Bool\(\) bool](#)
 [func \(v Value\) Bytes\(\) \[\]byte](#)
 [func \(v Value\) Call\(in \[\]Value\) \[\]Value](#)
 [func \(v Value\) CallSlice\(in \[\]Value\) \[\]Value](#)
 [func \(v Value\) CanAddr\(\) bool](#)
 [func \(v Value\) CanInterface\(\) bool](#)
 [func \(v Value\) CanSet\(\) bool](#)
 [func \(v Value\) Cap\(\) int](#)

[func \(v Value\) Close\(\)](#)
[func \(v Value\) Complex\(\) complex128](#)
[func \(v Value\) Elem\(\) Value](#)
[func \(v Value\) Field\(i int\) Value](#)
[func \(v Value\) FieldByIndex\(index \[\]int\) Value](#)
[func \(v Value\) FieldByName\(name string\) Value](#)
[func \(v Value\) FieldByNameFunc\(match func\(string\) bool\) Value](#)
[func \(v Value\) Float\(\) float64](#)
[func \(v Value\) Index\(i int\) Value](#)
[func \(v Value\) Int\(\) int64](#)
[func \(v Value\) Interface\(\) \(i interface{ }\)](#)
[func \(v Value\) InterfaceData\(\) \[2\]uintptr](#)
[func \(v Value\) IsNil\(\) bool](#)
[func \(v Value\) IsValid\(\) bool](#)
[func \(v Value\) Kind\(\) Kind](#)
[func \(v Value\) Len\(\) int](#)
[func \(v Value\) MapIndex\(key Value\) Value](#)
[func \(v Value\) MapKeys\(\) \[\]Value](#)
[func \(v Value\) Method\(i int\) Value](#)
[func \(v Value\) MethodByName\(name string\) Value](#)
[func \(v Value\) NumField\(\) int](#)
[func \(v Value\) NumMethod\(\) int](#)
[func \(v Value\) OverflowComplex\(x complex128\) bool](#)
[func \(v Value\) OverflowFloat\(x float64\) bool](#)
[func \(v Value\) OverflowInt\(x int64\) bool](#)
[func \(v Value\) OverflowUint\(x uint64\) bool](#)
[func \(v Value\) Pointer\(\) uintptr](#)
[func \(v Value\) Recv\(\) \(x Value, ok bool\)](#)
[func \(v Value\) Send\(x Value\)](#)
[func \(v Value\) Set\(x Value\)](#)
[func \(v Value\) SetBool\(x bool\)](#)
[func \(v Value\) SetBytes\(x \[\]byte\)](#)
[func \(v Value\) SetComplex\(x complex128\)](#)
[func \(v Value\) SetFloat\(x float64\)](#)
[func \(v Value\) SetInt\(x int64\)](#)
[func \(v Value\) SetLen\(n int\)](#)
[func \(v Value\) SetMapIndex\(key, val Value\)](#)
[func \(v Value\) SetPointer\(x unsafe.Pointer\)](#)
[func \(v Value\) SetString\(x string\)](#)

[func \(v Value\) SetUint\(x uint64\)](#)
[func \(v Value\) Slice\(beg, end int\) Value](#)
[func \(v Value\) String\(\) string](#)
[func \(v Value\) TryRecv\(\) \(x Value, ok bool\)](#)
[func \(v Value\) TrySend\(x Value\) bool](#)
[func \(v Value\) Type\(\) Type](#)
[func \(v Value\) Uint\(\) uint64](#)
[func \(v Value\) UnsafeAddr\(\) uintptr](#)
[type ValueError](#)
[func \(e *ValueError\) Error\(\) string](#)

Package files

[deepequal.go](#) [type.go](#) [value.go](#)

func Copy

```
func Copy(dst, src Value) int
```

Copy copies the contents of src into dst until either dst has been filled or src has been exhausted. It returns the number of elements copied. Dst and src each must have kind Slice or Array, and dst and src must have the same element type.

func [DeepEqual](#)

```
func DeepEqual(a1, a2 interface{}) bool
```

DeepEqual tests for deep equality. It uses normal == equality where possible but will scan members of arrays, slices, maps, and fields of structs. It correctly handles recursive types. Functions are equal only if they are both nil.

type [ChanDir](#)

```
type ChanDir int
```

ChanDir represents a channel type's direction.

```
const (  
    RecvDir ChanDir          = 1 << iota // <-chan  
    SendDir                  // chan<-  
    BothDir = RecvDir | SendDir // chan  
)
```

func (ChanDir) [String](#)

```
func (d ChanDir) String() string
```

type [Kind](#)

```
type Kind uint
```

A Kind represents the specific kind of type that a Type represents. The zero Kind is not a valid kind.

```
const (  
    Invalid Kind = iota  
    Bool  
    Int  
    Int8  
    Int16  
    Int32  
    Int64  
    Uint  
    Uint8  
    Uint16  
    Uint32  
    Uint64  
    Uintptr  
    Float32  
    Float64  
    Complex64  
    Complex128  
    Array  
    Chan  
    Func  
    Interface  
    Map  
    Ptr  
    Slice  
    String  
    Struct  
    UnsafePointer  
)
```

func (Kind) [String](#)

```
func (k Kind) String() string
```

type [Method](#)

```
type Method struct {
    // Name is the method name.
    // PkgPath is the package path that qualifies a lower case (unex
    // method name. It is empty for upper case (exported) method na
    // The combination of PkgPath and Name uniquely identifies a met
    // in a method set.
    // See http://golang.org/ref/spec#Uniqueness\_of\_identifiers
    Name    string
    PkgPath string

    Type Type // method type
    Func Value // func with receiver as first argument
    Index int    // index for Type.Method
}
```

Method represents a single method.

type [SliceHeader](#)

```
type SliceHeader struct {  
    Data uintptr  
    Len  int  
    Cap  int  
}
```

`SliceHeader` is the runtime representation of a slice. It cannot be used safely or portably.

type [StringHeader](#)

```
type StringHeader struct {  
    Data uintptr  
    Len  int  
}
```

StringHeader is the runtime representation of a string. It cannot be used safely or portably.

type StructField

```
type StructField struct {
    // Name is the field name.
    // PkgPath is the package path that qualifies a lower case (unex
    // field name. It is empty for upper case (exported) field name
    // See http://golang.org/ref/spec#Uniqueness\_of\_identifiers
    Name      string
    PkgPath   string

    Type      Type      // field type
    Tag       StructTag // field tag string
    Offset    uintptr  // offset within struct, in bytes
    Index     []int    // index sequence for Type.FieldByIndex
    Anonymous bool      // is an anonymous field
}
```

A StructField describes a single field in a struct.

type [StructTag](#)

```
type StructTag string
```

A StructTag is the tag string in a struct field.

By convention, tag strings are a concatenation of optionally space-separated key:"value" pairs. Each key is a non-empty string consisting of non-control characters other than space (U+0020 ' '), quote (U+0022 ""), and colon (U+003A ':'). Each value is quoted using U+0022 "" characters and Go string literal syntax.

func (**StructTag**) [Get](#)

```
func (tag StructTag) Get(key string) string
```

Get returns the value associated with key in the tag string. If there is no such key in the tag, Get returns the empty string. If the tag does not have the conventional format, the value returned by Get is unspecified.

type Type

```
type Type interface {

    // Align returns the alignment in bytes of a value of
    // this type when allocated in memory.
    Align() int

    // FieldAlign returns the alignment in bytes of a value of
    // this type when used as a field in a struct.
    FieldAlign() int

    // Method returns the i'th method in the type's method set.
    // It panics if i is not in the range [0, NumMethod()).
    //
    // For a non-interface type T or *T, the returned Method's Type
    // fields describe a function whose first argument is the receiver.
    //
    // For an interface type, the returned Method's Type field gives
    // method signature, without a receiver, and the Func field is nil.
    Method(int) Method

    // MethodByName returns the method with that name in the type's
    // method set and a boolean indicating if the method was found.
    //
    // For a non-interface type T or *T, the returned Method's Type
    // fields describe a function whose first argument is the receiver.
    //
    // For an interface type, the returned Method's Type field gives
    // method signature, without a receiver, and the Func field is nil.
    MethodByName(string) (Method, bool)

    // NumMethod returns the number of methods in the type's method
    // set.
    NumMethod() int

    // Name returns the type's name within its package.
    // It returns an empty string for unnamed types.
    Name() string

    // PkgPath returns a named type's package path, that is, the import
    // path that uniquely identifies the package, such as "encoding/base64".
    // If the type was predeclared (string, error) or unnamed (*T, slice),
    // the package path will be the empty string.
    PkgPath() string

    // Size returns the number of bytes needed to store
    // a value of the given type; it is analogous to unsafe.Sizeof.
```

```

Size() uintptr

// String returns a string representation of the type.
// The string representation may use shortened package names
// (e.g., base64 instead of "encoding/base64") and is not
// guaranteed to be unique among types. To test for equality,
// compare the Types directly.
String() string

// Kind returns the specific kind of this type.
Kind() Kind

// Implements returns true if the type implements the interface
Implements(u Type) bool

// AssignableTo returns true if a value of the type is assignabl
AssignableTo(u Type) bool

// Bits returns the size of the type in bits.
// It panics if the type's Kind is not one of the
// sized or unsized Int, Uint, Float, or Complex kinds.
Bits() int

// ChanDir returns a channel type's direction.
// It panics if the type's Kind is not Chan.
ChanDir() ChanDir

// IsVariadic returns true if a function type's final input para
// is a "... " parameter. If so, t.In(t.NumIn() - 1) returns the
// implicit actual type []T.
//
// For concreteness, if t represents func(x int, y ... float64),
//
// t.NumIn() == 2
// t.In(0) is the reflect.Type for "int"
// t.In(1) is the reflect.Type for "[]float64"
// t.IsVariadic() == true
//
// IsVariadic panics if the type's Kind is not Func.
IsVariadic() bool

// Elem returns a type's element type.
// It panics if the type's Kind is not Array, Chan, Map, Ptr, or
Elem() Type

// Field returns a struct type's i'th field.
// It panics if the type's Kind is not Struct.
// It panics if i is not in the range [0, NumField()).
Field(i int) StructField

```

```

// FieldByIndex returns the nested field corresponding
// to the index sequence. It is equivalent to calling Field
// successively for each index i.
// It panics if the type's Kind is not Struct.
FieldByIndex(index []int) StructField

// FieldByName returns the struct field with the given name
// and a boolean indicating if the field was found.
FieldByName(name string) (StructField, bool)

// FieldByNameFunc returns the first struct field with a name
// that satisfies the match function and a boolean indicating if
// the field was found.
FieldByNameFunc(match func(string) bool) (StructField, bool)

// In returns the type of a function type's i'th input parameter
// It panics if the type's Kind is not Func.
// It panics if i is not in the range [0, NumIn()).
In(i int) Type

// Key returns a map type's key type.
// It panics if the type's Kind is not Map.
Key() Type

// Len returns an array type's length.
// It panics if the type's Kind is not Array.
Len() int

// NumField returns a struct type's field count.
// It panics if the type's Kind is not Struct.
NumField() int

// NumIn returns a function type's input parameter count.
// It panics if the type's Kind is not Func.
NumIn() int

// NumOut returns a function type's output parameter count.
// It panics if the type's Kind is not Func.
NumOut() int

// Out returns the type of a function type's i'th output paramet
// It panics if the type's Kind is not Func.
// It panics if i is not in the range [0, NumOut()).
Out(i int) Type
// contains filtered or unexported methods
}

```

Type is the representation of a Go type.

Not all methods apply to all kinds of types. Restrictions, if any, are noted in the documentation for each method. Use the `Kind` method to find out the kind of type before calling kind-specific methods. Calling a method inappropriate to the kind of type causes a run-time panic.

func [PtrTo](#)

```
func PtrTo(t Type) Type
```

`PtrTo` returns the pointer type with element `t`. For example, if `t` represents type `Foo`, `PtrTo(t)` represents `*Foo`.

func [TypeOf](#)

```
func TypeOf(i interface{}) Type
```

`TypeOf` returns the reflection `Type` of the value in the `interface{}`. `TypeOf(nil)` returns `nil`.

type [Value](#)

```
type Value struct {  
    // contains filtered or unexported fields  
}
```

Value is the reflection interface to a Go value.

Not all methods apply to all kinds of values. Restrictions, if any, are noted in the documentation for each method. Use the Kind method to find out the kind of value before calling kind-specific methods. Calling a method inappropriate to the kind of type causes a run time panic.

The zero Value represents no value. Its IsValid method returns false, its Kind method returns Invalid, its String method returns "<invalid Value>", and all other methods panic. Most functions and methods never return an invalid value. If one does, its documentation states the conditions explicitly.

A Value can be used concurrently by multiple goroutines provided that the underlying Go value can be used concurrently for the equivalent direct operations.

func [Append](#)

```
func Append(s Value, x ...Value) Value
```

Append appends the values x to a slice s and returns the resulting slice. As in Go, each x's value must be assignable to the slice's element type.

func [AppendSlice](#)

```
func AppendSlice(s, t Value) Value
```

AppendSlice appends a slice t to a slice s and returns the resulting slice. The slices s and t must have the same element type.

func [Indirect](#)

```
func Indirect(v Value) Value
```

Indirect returns the value that v points to. If v is a nil pointer, Indirect returns a zero Value. If v is not a pointer, Indirect returns v.

func [MakeChan](#)

```
func MakeChan(typ Type, buffer int) Value
```

MakeChan creates a new channel with the specified type and buffer size.

func [MakeMap](#)

```
func MakeMap(typ Type) Value
```

MakeMap creates a new map of the specified type.

func [MakeSlice](#)

```
func MakeSlice(typ Type, len, cap int) Value
```

MakeSlice creates a new zero-initialized slice value for the specified slice type, length, and capacity.

func [New](#)

```
func New(typ Type) Value
```

New returns a Value representing a pointer to a new zero value for the specified type. That is, the returned Value's Type is PtrTo(t).

func [NewAt](#)

```
func NewAt(typ Type, p unsafe.Pointer) Value
```

NewAt returns a Value representing a pointer to a value of the specified type, using p as that pointer.

func [ValueOf](#)

```
func ValueOf(i interface{}) Value
```

ValueOf returns a new Value initialized to the concrete value stored in the interface i. ValueOf(nil) returns the zero Value.

func [Zero](#)

```
func Zero(typ Type) Value
```

Zero returns a Value representing a zero value for the specified type. The result is different from the zero value of the Value struct, which represents no value at all. For example, Zero(.TypeOf(42)) returns a Value with Kind Int and value 0.

func (Value) [Addr](#)

```
func (v Value) Addr() Value
```

Addr returns a pointer value representing the address of v. It panics if CanAddr() returns false. Addr is typically used to obtain a pointer to a struct field or slice element in order to call a method that requires a pointer receiver.

func (Value) [Bool](#)

```
func (v Value) Bool() bool
```

Bool returns v's underlying value. It panics if v's kind is not Bool.

func (Value) [Bytes](#)

```
func (v Value) Bytes() []byte
```

Bytes returns v's underlying value. It panics if v's underlying value is not a slice of bytes.

func (Value) [Call](#)

```
func (v Value) Call(in []Value) []Value
```

Call calls the function v with the input arguments in. For example, if len(in) == 3, v.Call(in) represents the Go call v(in[0], in[1], in[2]). Call panics if v's Kind is not Func. It returns the output results as Values. As in Go, each input argument must be assignable to the type of the function's corresponding input parameter. If

v is a variadic function, Call creates the variadic slice parameter itself, copying in the corresponding values.

func (Value) [CallSlice](#)

```
func (v Value) CallSlice(in []Value) []Value
```

CallSlice calls the variadic function v with the input arguments in, assigning the slice in[len(in)-1] to v's final variadic argument. For example, if len(in) == 3, v.Call(in) represents the Go call v(in[0], in[1], in[2]...). Call panics if v's Kind is not Func or if v is not variadic. It returns the output results as Values. As in Go, each input argument must be assignable to the type of the function's corresponding input parameter.

func (Value) [CanAddr](#)

```
func (v Value) CanAddr() bool
```

CanAddr returns true if the value's address can be obtained with Addr. Such values are called addressable. A value is addressable if it is an element of a slice, an element of an addressable array, a field of an addressable struct, or the result of dereferencing a pointer. If CanAddr returns false, calling Addr will panic.

func (Value) [CanInterface](#)

```
func (v Value) CanInterface() bool
```

CanInterface returns true if Interface can be used without panicking.

func (Value) [CanSet](#)

```
func (v Value) CanSet() bool
```

CanSet returns true if the value of v can be changed. A Value can be changed only if it is addressable and was not obtained by the use of unexported struct fields. If CanSet returns false, calling Set or any type-specific setter (e.g., SetBool, SetInt64) will panic.

func (Value) [Cap](#)

```
func (v Value) Cap() int
```

Cap returns v's capacity. It panics if v's Kind is not Array, Chan, or Slice.

func (Value) [Close](#)

```
func (v Value) Close()
```

Close closes the channel v. It panics if v's Kind is not Chan.

func (Value) [Complex](#)

```
func (v Value) Complex() complex128
```

Complex returns v's underlying value, as a complex128. It panics if v's Kind is not Complex64 or Complex128

func (Value) [Elem](#)

```
func (v Value) Elem() Value
```

Elem returns the value that the interface v contains or that the pointer v points to. It panics if v's Kind is not Interface or Ptr. It returns the zero Value if v is nil.

func (Value) [Field](#)

```
func (v Value) Field(i int) Value
```

Field returns the i'th field of the struct v. It panics if v's Kind is not Struct or i is out of range.

func (Value) [FieldByIndex](#)

```
func (v Value) FieldByIndex(index []int) Value
```

FieldByIndex returns the nested field corresponding to index. It panics if v's Kind is not struct.

func (Value) [FieldByName](#)

```
func (v Value) FieldByName(name string) Value
```

FieldByName returns the struct field with the given name. It returns the zero Value if no field was found. It panics if v's Kind is not struct.

func (Value) [FieldByNameFunc](#)

```
func (v Value) FieldByNameFunc(match func(string) bool) Value
```

FieldByNameFunc returns the struct field with a name that satisfies the match function. It panics if v's Kind is not struct. It returns the zero Value if no field was found.

func (Value) [Float](#)

```
func (v Value) Float() float64
```

Float returns v's underlying value, as a float64. It panics if v's Kind is not Float32 or Float64

func (Value) [Index](#)

```
func (v Value) Index(i int) Value
```

Index returns v's i'th element. It panics if v's Kind is not Array or Slice or i is out of range.

func (Value) [Int](#)

```
func (v Value) Int() int64
```

Int returns v's underlying value, as an int64. It panics if v's Kind is not Int, Int8, Int16, Int32, or Int64.

func (Value) [Interface](#)

```
func (v Value) Interface() (i interface{})
```

Interface returns v's current value as an interface{}. It is equivalent to:

```
var i interface{} = (v's underlying value)
```

If `v` is a method obtained by invoking `Value.Method` (as opposed to `Type.Method`), `Interface` cannot return an interface value, so it panics. It also panics if the `Value` was obtained by accessing unexported struct fields.

func (Value) [InterfaceData](#)

```
func (v Value) InterfaceData() [2]uintptr
```

`InterfaceData` returns the interface `v`'s value as a `uintptr` pair. It panics if `v`'s `Kind` is not `Interface`.

func (Value) [IsNil](#)

```
func (v Value) IsNil() bool
```

`IsNil` returns true if `v` is a nil value. It panics if `v`'s `Kind` is not `Chan`, `Func`, `Interface`, `Map`, `Ptr`, or `Slice`.

func (Value) [IsValid](#)

```
func (v Value) IsValid() bool
```

`IsValid` returns true if `v` represents a value. It returns false if `v` is the zero `Value`. If `IsValid` returns false, all other methods except `String` panic. Most functions and methods never return an invalid value. If one does, its documentation states the conditions explicitly.

func (Value) [Kind](#)

```
func (v Value) Kind() Kind
```

`Kind` returns `v`'s `Kind`. If `v` is the zero `Value` (`IsValid` returns false), `Kind` returns `Invalid`.

func (Value) [Len](#)

```
func (v Value) Len() int
```

Len returns v's length. It panics if v's Kind is not Array, Chan, Map, Slice, or String.

func (Value) [MapIndex](#)

```
func (v Value) MapIndex(key Value) Value
```

MapIndex returns the value associated with key in the map v. It panics if v's Kind is not Map. It returns the zero Value if key is not found in the map or if v represents a nil map. As in Go, the key's value must be assignable to the map's key type.

func (Value) [MapKeys](#)

```
func (v Value) MapKeys() []Value
```

MapKeys returns a slice containing all the keys present in the map, in unspecified order. It panics if v's Kind is not Map. It returns an empty slice if v represents a nil map.

func (Value) [Method](#)

```
func (v Value) Method(i int) Value
```

Method returns a function value corresponding to v's i'th method. The arguments to a Call on the returned function should not include a receiver; the returned function will always use v as the receiver. Method panics if i is out of range.

func (Value) [MethodByName](#)

```
func (v Value) MethodByName(name string) Value
```

MethodByName returns a function value corresponding to the method of v with the given name. The arguments to a Call on the returned function should not include a receiver; the returned function will always use v as the receiver. It returns the zero Value if no method was found.

func (Value) [NumField](#)

```
func (v Value) NumField() int
```

NumField returns the number of fields in the struct v. It panics if v's Kind is not Struct.

func (Value) [NumMethod](#)

```
func (v Value) NumMethod() int
```

NumMethod returns the number of methods in the value's method set.

func (Value) [OverflowComplex](#)

```
func (v Value) OverflowComplex(x complex128) bool
```

OverflowComplex returns true if the complex128 x cannot be represented by v's type. It panics if v's Kind is not Complex64 or Complex128.

func (Value) [OverflowFloat](#)

```
func (v Value) OverflowFloat(x float64) bool
```

OverflowFloat returns true if the float64 x cannot be represented by v's type. It panics if v's Kind is not Float32 or Float64.

func (Value) [OverflowInt](#)

```
func (v Value) OverflowInt(x int64) bool
```

OverflowInt returns true if the int64 x cannot be represented by v's type. It panics if v's Kind is not Int, Int8, int16, Int32, or Int64.

func (Value) [OverflowUint](#)

```
func (v Value) OverflowUint(x uint64) bool
```

OverflowUint returns true if the uint64 x cannot be represented by v's type. It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16, Uint32, or Uint64.

func (Value) [Pointer](#)

```
func (v Value) Pointer() uintptr
```

Pointer returns v's value as a uintptr. It returns uintptr instead of unsafe.Pointer so that code using reflect cannot obtain unsafe.Pointers without importing the unsafe package explicitly. It panics if v's Kind is not Chan, Func, Map, Ptr, Slice, or UnsafePointer.

func (Value) [Recv](#)

```
func (v Value) Recv() (x Value, ok bool)
```

Recv receives and returns a value from the channel v. It panics if v's Kind is not Chan. The receive blocks until a value is ready. The boolean value ok is true if the value x corresponds to a send on the channel, false if it is a zero value received because the channel is closed.

func (Value) [Send](#)

```
func (v Value) Send(x Value)
```

Send sends x on the channel v. It panics if v's kind is not Chan or if x's type is not the same type as v's element type. As in Go, x's value must be assignable to the channel's element type.

func (Value) [Set](#)

```
func (v Value) Set(x Value)
```

Set assigns x to the value v. It panics if CanSet returns false. As in Go, x's value must be assignable to v's type.

func (Value) [SetBool](#)

```
func (v Value) SetBool(x bool)
```

SetBool sets v's underlying value. It panics if v's Kind is not Bool or if CanSet() is false.

func (Value) [SetBytes](#)

```
func (v Value) SetBytes(x []byte)
```

SetBytes sets v's underlying value. It panics if v's underlying value is not a slice of bytes.

func (Value) [SetComplex](#)

```
func (v Value) SetComplex(x complex128)
```

SetComplex sets v's underlying value to x. It panics if v's Kind is not Complex64 or Complex128, or if CanSet() is false.

func (Value) [SetFloat](#)

```
func (v Value) SetFloat(x float64)
```

SetFloat sets v's underlying value to x. It panics if v's Kind is not Float32 or Float64, or if CanSet() is false.

func (Value) [SetInt](#)

```
func (v Value) SetInt(x int64)
```

SetInt sets v's underlying value to x. It panics if v's Kind is not Int, Int8, Int16, Int32, or Int64, or if CanSet() is false.

func (Value) [SetLen](#)

```
func (v Value) SetLen(n int)
```

SetLen sets v's length to n. It panics if v's Kind is not Slice or if n is negative or greater than the capacity of the slice.

func (Value) [SetMapIndex](#)

```
func (v Value) SetMapIndex(key, val Value)
```

SetMapIndex sets the value associated with key in the map v to val. It panics if v's Kind is not Map. If val is the zero Value, SetMapIndex deletes the key from the map. As in Go, key's value must be assignable to the map's key type, and val's value must be assignable to the map's value type.

func (Value) [SetPointer](#)

```
func (v Value) SetPointer(x unsafe.Pointer)
```

SetPointer sets the unsafe.Pointer value v to x. It panics if v's Kind is not UnsafePointer.

func (Value) [SetString](#)

```
func (v Value) SetString(x string)
```

SetString sets v's underlying value to x. It panics if v's Kind is not String or if CanSet() is false.

func (Value) [SetUint](#)

```
func (v Value) SetUint(x uint64)
```

SetUint sets v's underlying value to x. It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16, Uint32, or Uint64, or if CanSet() is false.

func (Value) [Slice](#)

```
func (v Value) Slice(beg, end int) Value
```

Slice returns a slice of v. It panics if v's Kind is not Array or Slice.

func (Value) [String](#)

```
func (v Value) String() string
```

String returns the string v's underlying value, as a string. String is a special case because of Go's String method convention. Unlike the other getters, it does not panic if v's Kind is not String. Instead, it returns a string of the form "<T value>" where T is v's type.

func (Value) [TryRecv](#)

```
func (v Value) TryRecv() (x Value, ok bool)
```

TryRecv attempts to receive a value from the channel v but will not block. It panics if v's Kind is not Chan. If the receive cannot finish without blocking, x is the zero Value. The boolean ok is true if the value x corresponds to a send on the channel, false if it is a zero value received because the channel is closed.

func (Value) [TrySend](#)

```
func (v Value) TrySend(x Value) bool
```

TrySend attempts to send x on the channel v but will not block. It panics if v's Kind is not Chan. It returns true if the value was sent, false otherwise. As in Go, x's value must be assignable to the channel's element type.

func (Value) [Type](#)

```
func (v Value) Type() Type
```

Type returns v's type.

func (Value) [Uint](#)

```
func (v Value) Uint() uint64
```

Uint returns v's underlying value, as a uint64. It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16, Uint32, or Uint64.

func (Value) [UnsafeAddr](#)

```
func (v Value) UnsafeAddr() uintptr
```

UnsafeAddr returns a pointer to v's data. It is for advanced clients that also import the "unsafe" package. It panics if v is not addressable.

type [ValueError](#)

```
type ValueError struct {  
    Method string  
    Kind    Kind  
}
```

A `ValueError` occurs when a `Value` method is invoked on a `Value` that does not support it. Such cases are documented in the description of each method.

func (*[ValueError](#)) [Error](#)

```
func (e *ValueError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package regexp

```
import "regexp"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package `regexp` implements regular expression search.

The syntax of the regular expressions accepted is the same general syntax used by Perl, Python, and other languages. More precisely, it is the syntax accepted by RE2 and described at <http://code.google.com/p/re2/wiki/Syntax>, except for `\C`.

All characters are UTF-8-encoded code points.

There are 16 methods of `Regexp` that match a regular expression and identify the matched text. Their names are matched by this regular expression:

```
Find(All)?(String)?(Submatch)?(Index)?
```

If `'All'` is present, the routine matches successive non-overlapping matches of the entire expression. Empty matches abutting a preceding match are ignored. The return value is a slice containing the successive return values of the corresponding non-`'All'` routine. These routines take an extra integer argument, `n`; if `n >= 0`, the function returns at most `n` matches/submatches.

If `'String'` is present, the argument is a string; otherwise it is a slice of bytes; return values are adjusted as appropriate.

If `'Submatch'` is present, the return value is a slice identifying the successive submatches of the expression. Submatches are matches of parenthesized subexpressions within the regular expression, numbered from left to right in order of opening parenthesis. Submatch 0 is the match of the entire expression, submatch 1 the match of the first parenthesized subexpression, and so on.

If `'Index'` is present, matches and submatches are identified by byte index pairs within the input string: `result[2*n:2*n+1]` identifies the indexes of the `n`th submatch. The pair for `n==0` identifies the match of the entire expression. If `'Index'` is not present, the match is identified by the text of the match/submatch. If an index is negative, it means that subexpression did not match any string in the input.

There is also a subset of the methods that can be applied to text read from a `RuneReader`:

MatchReader, FindReaderIndex, FindReaderSubmatchIndex

This set may grow. Note that regular expression matches may need to examine text beyond the text returned by a match, so the methods that match text from a RuneReader may read arbitrarily far into the input before returning.

(There are a few other methods that do not match this pattern.)

Index

[func Match\(pattern string, b \[\]byte\) \(matched bool, error error\)](#)
[func MatchReader\(pattern string, r io.RuneReader\) \(matched bool, error error\)](#)
[func MatchString\(pattern string, s string\) \(matched bool, error error\)](#)
[func QuoteMeta\(s string\) string](#)
[type Regexp](#)
 [func Compile\(expr string\) \(*Regexp, error\)](#)
 [func CompilePOSIX\(expr string\) \(*Regexp, error\)](#)
 [func MustCompile\(str string\) *Regexp](#)
 [func MustCompilePOSIX\(str string\) *Regexp](#)
 [func \(re *Regexp\) Expand\(dst \[\]byte, template \[\]byte, src \[\]byte, match \[\]int\) \[\]byte](#)
 [func \(re *Regexp\) ExpandString\(dst \[\]byte, template string, src string, match \[\]int\) \[\]byte](#)
 [func \(re *Regexp\) Find\(b \[\]byte\) \[\]byte](#)
 [func \(re *Regexp\) FindAll\(b \[\]byte, n int\) \[\]\[\]byte](#)
 [func \(re *Regexp\) FindAllIndex\(b \[\]byte, n int\) \[\]\[\]int](#)
 [func \(re *Regexp\) FindAllString\(s string, n int\) \[\]string](#)
 [func \(re *Regexp\) FindAllStringIndex\(s string, n int\) \[\]\[\]int](#)
 [func \(re *Regexp\) FindAllStringSubmatch\(s string, n int\) \[\]\[\]string](#)
 [func \(re *Regexp\) FindAllStringSubmatchIndex\(s string, n int\) \[\]\[\]int](#)
 [func \(re *Regexp\) FindAllSubmatch\(b \[\]byte, n int\) \[\]\[\]\[\]byte](#)
 [func \(re *Regexp\) FindAllSubmatchIndex\(b \[\]byte, n int\) \[\]\[\]int](#)
 [func \(re *Regexp\) FindIndex\(b \[\]byte\) \(loc \[\]int\)](#)
 [func \(re *Regexp\) FindReaderIndex\(r io.RuneReader\) \(loc \[\]int\)](#)
 [func \(re *Regexp\) FindReaderSubmatchIndex\(r io.RuneReader\) \[\]int](#)
 [func \(re *Regexp\) FindString\(s string\) string](#)
 [func \(re *Regexp\) FindStringIndex\(s string\) \(loc \[\]int\)](#)
 [func \(re *Regexp\) FindStringSubmatch\(s string\) \[\]string](#)
 [func \(re *Regexp\) FindStringSubmatchIndex\(s string\) \[\]int](#)
 [func \(re *Regexp\) FindSubmatch\(b \[\]byte\) \[\]\[\]byte](#)
 [func \(re *Regexp\) FindSubmatchIndex\(b \[\]byte\) \[\]int](#)
 [func \(re *Regexp\) LiteralPrefix\(\) \(prefix string, complete bool\)](#)
 [func \(re *Regexp\) Match\(b \[\]byte\) bool](#)
 [func \(re *Regexp\) MatchReader\(r io.RuneReader\) bool](#)

[func \(re *Regexp\) MatchString\(s string\) bool](#)
[func \(re *Regexp\) NumSubexp\(\) int](#)
[func \(re *Regexp\) ReplaceAll\(src, repl \[\]byte\) \[\]byte](#)
[func \(re *Regexp\) ReplaceAllFunc\(src \[\]byte, repl func\(\[\]byte\) \[\]byte\) \[\]byte](#)
[\[\]byte](#)
[func \(re *Regexp\) ReplaceAllLiteral\(src, repl \[\]byte\) \[\]byte](#)
[func \(re *Regexp\) ReplaceAllLiteralString\(src, repl string\) string](#)
[func \(re *Regexp\) ReplaceAllString\(src, repl string\) string](#)
[func \(re *Regexp\) ReplaceAllStringFunc\(src string, repl func\(string\) string\) string](#)
[string\) string](#)
[func \(re *Regexp\) String\(\) string](#)
[func \(re *Regexp\) SubexpNames\(\) \[\]string](#)

Package files

exec.go regex.go

func [Match](#)

```
func Match(pattern string, b []byte) (matched bool, error error)
```

Match checks whether a textual regular expression matches a byte slice. More complicated queries need to use `Compile` and the full `Regexp` interface.

func [MatchReader](#)

```
func MatchReader(pattern string, r io.RuneReader) (matched bool, err
```

MatchReader checks whether a textual regular expression matches the text read by the RuneReader. More complicated queries need to use Compile and the full Regexp interface.

func [MatchString](#)

`func MatchString(pattern string, s string) (matched bool, error error)`

MatchString checks whether a textual regular expression matches a string. More complicated queries need to use Compile and the full Regexp interface.

func [QuoteMeta](#)

```
func QuoteMeta(s string) string
```

QuoteMeta returns a string that quotes all regular expression metacharacters inside the argument text; the returned string is a regular expression matching the literal text. For example, `QuoteMeta(`[foo]`)` returns ``\[foo\]``.

type [Regexp](#)

```
type Regexp struct {  
    // contains filtered or unexported fields  
}
```

Regexp is the representation of a compiled regular expression. The public interface is entirely through methods. A Regexp is safe for concurrent use by multiple goroutines.

func [Compile](#)

```
func Compile(expr string) (*Regexp, error)
```

Compile parses a regular expression and returns, if successful, a Regexp object that can be used to match against text.

When matching against text, the regexp returns a match that begins as early as possible in the input (leftmost), and among those it chooses the one that a backtracking search would have found first. This so-called leftmost-first matching is the same semantics that Perl, Python, and other implementations use, although this package implements it without the expense of backtracking. For POSIX leftmost-longest matching, see [CompilePOSIX](#).

func [CompilePOSIX](#)

```
func CompilePOSIX(expr string) (*Regexp, error)
```

CompilePOSIX is like [Compile](#) but restricts the regular expression to POSIX ERE (egrep) syntax and changes the match semantics to leftmost-longest.

That is, when matching against text, the regexp returns a match that begins as early as possible in the input (leftmost), and among those it chooses a match that is as long as possible. This so-called leftmost-longest matching is the same semantics that early regular expression implementations used and that POSIX specifies.

However, there can be multiple leftmost-longest matches, with different submatch choices, and here this package diverges from POSIX. Among the

possible leftmost-longest matches, this package chooses the one that a backtracking search would have found first, while POSIX specifies that the match be chosen to maximize the length of the first subexpression, then the second, and so on from left to right. The POSIX rule is computationally prohibitive and not even well-defined. See <http://swtch.com/~rsc/regexp/regexp2.html#posix> for details.

func [MustCompile](#)

```
func MustCompile(str string) *Regexp
```

MustCompile is like Compile but panics if the expression cannot be parsed. It simplifies safe initialization of global variables holding compiled regular expressions.

func [MustCompilePOSIX](#)

```
func MustCompilePOSIX(str string) *Regexp
```

MustCompilePOSIX is like CompilePOSIX but panics if the expression cannot be parsed. It simplifies safe initialization of global variables holding compiled regular expressions.

func (*Regexp) [Expand](#)

```
func (re *Regexp) Expand(dst []byte, template []byte, src []byte, ma
```

Expand appends template to dst and returns the result; during the append, Expand replaces variables in the template with corresponding matches drawn from src. The match slice should have been returned by FindSubmatchIndex.

In the template, a variable is denoted by a substring of the form \$name or \${name}, where name is a non-empty sequence of letters, digits, and underscores. A purely numeric name like \$1 refers to the submatch with the corresponding index; other names refer to capturing parentheses named with the (?P<name>...) syntax. A reference to an out of range or unmatched index or a name that is not present in the regular expression is replaced with an empty string.

In the \$name form, name is taken to be as long as possible: \$1x is equivalent to

`${1x}`, not `${1}x`, and, `$10` is equivalent to `${10}`, not `${1}0`.

To insert a literal `$` in the output, use `$$` in the template.

func (*Regex) [ExpandString](#)

```
func (re *Regex) ExpandString(dst []byte, template string, src stri
```

`ExpandString` is like `Expand` but the template and source are strings. It appends to and returns a byte slice in order to give the calling code control over allocation.

func (*Regex) [Find](#)

```
func (re *Regex) Find(b []byte) []byte
```

`Find` returns a slice holding the text of the leftmost match in `b` of the regular expression. A return value of `nil` indicates no match.

func (*Regex) [FindAll](#)

```
func (re *Regex) FindAll(b []byte, n int) [][]byte
```

`FindAll` is the 'All' version of `Find`; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of `nil` indicates no match.

func (*Regex) [FindAllIndex](#)

```
func (re *Regex) FindAllIndex(b []byte, n int) [][]int
```

`FindAllIndex` is the 'All' version of `FindIndex`; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of `nil` indicates no match.

func (*Regex) [FindAllString](#)

```
func (re *Regex) FindAllString(s string, n int) []string
```

`FindAllString` is the 'All' version of `FindString`; it returns a slice of all successive

matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

func (*Regex) [FindAllStringIndex](#)

```
func (re *Regex) FindAllStringIndex(s string, n int) [][]int
```

FindAllStringIndex is the 'All' version of FindStringIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

func (*Regex) [FindAllStringSubmatch](#)

```
func (re *Regex) FindAllStringSubmatch(s string, n int) [][]string
```

FindAllStringSubmatch is the 'All' version of FindStringSubmatch; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

func (*Regex) [FindAllStringSubmatchIndex](#)

```
func (re *Regex) FindAllStringSubmatchIndex(s string, n int) [][]int
```

FindAllStringSubmatchIndex is the 'All' version of FindStringSubmatchIndex; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

func (*Regex) [FindAllSubmatch](#)

```
func (re *Regex) FindAllSubmatch(b []byte, n int) [][][]byte
```

FindAllSubmatch is the 'All' version of FindSubmatch; it returns a slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

func (*Regex) [FindAllSubmatchIndex](#)

```
func (re *Regex) FindAllSubmatchIndex(b []byte, n int) [][]int
```

FindAllSubmatchIndex is the 'All' version of FindSubmatchIndex; it returns a

slice of all successive matches of the expression, as defined by the 'All' description in the package comment. A return value of nil indicates no match.

func (*Regex) [FindIndex](#)

```
func (re *Regex) FindIndex(b []byte) (loc []int)
```

FindIndex returns a two-element slice of integers defining the location of the leftmost match in b of the regular expression. The match itself is at b[loc[0]:loc[1]]. A return value of nil indicates no match.

func (*Regex) [FindReaderIndex](#)

```
func (re *Regex) FindReaderIndex(r io.RuneReader) (loc []int)
```

FindReaderIndex returns a two-element slice of integers defining the location of the leftmost match of the regular expression in text read from the RuneReader. The match itself is at s[loc[0]:loc[1]]. A return value of nil indicates no match.

func (*Regex) [FindReaderSubmatchIndex](#)

```
func (re *Regex) FindReaderSubmatchIndex(r io.RuneReader) []int
```

FindReaderSubmatchIndex returns a slice holding the index pairs identifying the leftmost match of the regular expression of text read by the RuneReader, and the matches, if any, of its subexpressions, as defined by the 'Submatch' and 'Index' descriptions in the package comment. A return value of nil indicates no match.

func (*Regex) [FindString](#)

```
func (re *Regex) FindString(s string) string
```

FindString returns a string holding the text of the leftmost match in s of the regular expression. If there is no match, the return value is an empty string, but it will also be empty if the regular expression successfully matches an empty string. Use FindStringIndex or FindStringSubmatch if it is necessary to distinguish these cases.

func (*Regex) [FindStringIndex](#)

```
func (re *Regexp) FindStringIndex(s string) (loc []int)
```

FindStringIndex returns a two-element slice of integers defining the location of the leftmost match in s of the regular expression. The match itself is at s[loc[0]:loc[1]]. A return value of nil indicates no match.

func (*Regexp) [FindStringSubmatch](#)

```
func (re *Regexp) FindStringSubmatch(s string) []string
```

FindStringSubmatch returns a slice of strings holding the text of the leftmost match of the regular expression in s and the matches, if any, of its subexpressions, as defined by the 'Submatch' description in the package comment. A return value of nil indicates no match.

func (*Regexp) [FindStringSubmatchIndex](#)

```
func (re *Regexp) FindStringSubmatchIndex(s string) []int
```

FindStringSubmatchIndex returns a slice holding the index pairs identifying the leftmost match of the regular expression in s and the matches, if any, of its subexpressions, as defined by the 'Submatch' and 'Index' descriptions in the package comment. A return value of nil indicates no match.

func (*Regexp) [FindSubmatch](#)

```
func (re *Regexp) FindSubmatch(b []byte) [][]byte
```

FindSubmatch returns a slice of slices holding the text of the leftmost match of the regular expression in b and the matches, if any, of its subexpressions, as defined by the 'Submatch' descriptions in the package comment. A return value of nil indicates no match.

func (*Regexp) [FindSubmatchIndex](#)

```
func (re *Regexp) FindSubmatchIndex(b []byte) []int
```

FindSubmatchIndex returns a slice holding the index pairs identifying the leftmost match of the regular expression in b and the matches, if any, of its subexpressions, as defined by the 'Submatch' and 'Index' descriptions in the

package comment. A return value of nil indicates no match.

func (*Regexp) [LiteralPrefix](#)

```
func (re *Regexp) LiteralPrefix() (prefix string, complete bool)
```

LiteralPrefix returns a literal string that must begin any match of the regular expression re. It returns the boolean true if the literal string comprises the entire regular expression.

func (*Regexp) [Match](#)

```
func (re *Regexp) Match(b []byte) bool
```

Match returns whether the Regexp matches the byte slice b. The return value is a boolean: true for match, false for no match.

func (*Regexp) [MatchReader](#)

```
func (re *Regexp) MatchReader(r io.RuneReader) bool
```

MatchReader returns whether the Regexp matches the text read by the RuneReader. The return value is a boolean: true for match, false for no match.

func (*Regexp) [MatchString](#)

```
func (re *Regexp) MatchString(s string) bool
```

MatchString returns whether the Regexp matches the string s. The return value is a boolean: true for match, false for no match.

func (*Regexp) [NumSubexp](#)

```
func (re *Regexp) NumSubexp() int
```

NumSubexp returns the number of parenthesized subexpressions in this Regexp.

func (*Regexp) [ReplaceAll](#)

```
func (re *Regexp) ReplaceAll(src, repl []byte) []byte
```

ReplaceAll returns a copy of src, replacing matches of the Regexp with the replacement string repl. Inside repl, \$ signs are interpreted as in Expand, so for instance \$1 represents the text of the first submatch.

func (*Regexp) [ReplaceAllFunc](#)

```
func (re *Regexp) ReplaceAllFunc(src []byte, repl func([]byte) []byte)
```

ReplaceAllFunc returns a copy of src in which all matches of the Regexp have been replaced by the return value of of function repl applied to the matched byte slice. The replacement returned by repl is substituted directly, without using Expand.

func (*Regexp) [ReplaceAllLiteral](#)

```
func (re *Regexp) ReplaceAllLiteral(src, repl []byte) []byte
```

ReplaceAllLiteral returns a copy of src, replacing matches of the Regexp with the replacement bytes repl. The replacement repl is substituted directly, without using Expand.

func (*Regexp) [ReplaceAllLiteralString](#)

```
func (re *Regexp) ReplaceAllLiteralString(src, repl string) string
```

ReplaceAllStringLiteral returns a copy of src, replacing matches of the Regexp with the replacement string repl. The replacement repl is substituted directly, without using Expand.

func (*Regexp) [ReplaceAllString](#)

```
func (re *Regexp) ReplaceAllString(src, repl string) string
```

ReplaceAllString returns a copy of src, replacing matches of the Regexp with the replacement string repl. Inside repl, \$ signs are interpreted as in Expand, so for instance \$1 represents the text of the first submatch.

func (*Regexp) [ReplaceAllStringFunc](#)

```
func (re *Regexp) ReplaceAllStringFunc(src string, repl func(string))
```

ReplaceAllStringFunc returns a copy of src in which all matches of the Regexp have been replaced by the return value of of function repl applied to the matched substring. The replacement returned by repl is substituted directly, without using Expand.

func (*Regexp) [String](#)

```
func (re *Regexp) String() string
```

String returns the source text used to compile the regular expression.

func (*Regexp) [SubexpNames](#)

```
func (re *Regexp) SubexpNames() []string
```

SubexpNames returns the names of the parenthesized subexpressions in this Regexp. The name for the first sub-expression is names[1], so that if m is a match slice, the name for m[i] is SubexpNames()[i]. Since the Regexp as a whole cannot be named, names[0] is always the empty string. The slice should not be modified.

Subdirectories

Name **Synopsis**

[syntax](#) Package syntax parses regular expressions into parse trees and compiles parse trees into programs.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package syntax

```
import "regexp/syntax"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `syntax` parses regular expressions into parse trees and compiles parse trees into programs. Most clients of regular expressions will use the facilities of package `regexp` (such as `Compile` and `Match`) instead of this package.

Index

[func IsWordChar\(r rune\) bool](#)
[type EmptyOp](#)
 [func EmptyOpContext\(r1, r2 rune\) EmptyOp](#)
[type Error](#)
 [func \(e *Error\) Error\(\) string](#)
[type ErrorCode](#)
 [func \(e ErrorCode\) String\(\) string](#)
[type Flags](#)
[type Inst](#)
 [func \(i *Inst\) MatchEmptyWidth\(before rune, after rune\) bool](#)
 [func \(i *Inst\) MatchRune\(r rune\) bool](#)
 [func \(i *Inst\) String\(\) string](#)
[type InstOp](#)
[type Op](#)
[type Prog](#)
 [func Compile\(re *Regexp\) \(*Prog, error\)](#)
 [func \(p *Prog\) Prefix\(\) \(prefix string, complete bool\)](#)
 [func \(p *Prog\) StartCond\(\) EmptyOp](#)
 [func \(p *Prog\) String\(\) string](#)
[type Regexp](#)
 [func Parse\(s string, flags Flags\) \(*Regexp, error\)](#)
 [func \(re *Regexp\) CapNames\(\) \[\]string](#)
 [func \(x *Regexp\) Equal\(y *Regexp\) bool](#)
 [func \(re *Regexp\) MaxCap\(\) int](#)
 [func \(re *Regexp\) Simplify\(\) *Regexp](#)
 [func \(re *Regexp\) String\(\) string](#)

Package files

[compile.go](#) [parse.go](#) [perl_groups.go](#) [prog.go](#) [regexp.go](#) [simplify.go](#)

func IsWordChar

```
func IsWordChar(r rune) bool
```

IsWordChar reports whether r is consider a “word character” during the evaluation of the \b and \B zero-width assertions. These assertions are ASCII-only: the word characters are [A-Za-z0-9_].

type [EmptyOp](#)

```
type EmptyOp uint8
```

An EmptyOp specifies a kind or mixture of zero-width assertions.

```
const (  
    EmptyBeginLine EmptyOp = 1 << iota  
    EmptyEndLine  
    EmptyBeginText  
    EmptyEndText  
    EmptyWordBoundary  
    EmptyNoWordBoundary  
)
```

func [EmptyOpContext](#)

```
func EmptyOpContext(r1, r2 rune) EmptyOp
```

EmptyOpContext returns the zero-width assertions satisfied at the position between the runes r1 and r2. Passing r1 == -1 indicates that the position is at the beginning of the text. Passing r2 == -1 indicates that the position is at the end of the text.

type [Error](#)

```
type Error struct {  
    Code ErrorCode  
    Expr string  
}
```

An Error describes a failure to parse a regular expression and gives the offending expression.

func (***Error**) [Error](#)

```
func (e *Error) Error() string
```

type [ErrorCode](#)

```
type ErrorCode string
```

An ErrorCode describes a failure to parse a regular expression.

```
const (  
    // Unexpected error  
    ErrInternalError ErrorCode = "regexp/syntax: internal error"  
  
    // Parse errors  
    ErrInvalidCharClass      ErrorCode = "invalid character class"  
    ErrInvalidCharRange     ErrorCode = "invalid character class ra  
    ErrInvalidEscape         ErrorCode = "invalid escape sequence"  
    ErrInvalidNamedCapture  ErrorCode = "invalid named capture"  
    ErrInvalidPerlOp        ErrorCode = "invalid or unsupported Per  
    ErrInvalidRepeatOp      ErrorCode = "invalid nested repetition"  
    ErrInvalidRepeatSize    ErrorCode = "invalid repeat count"  
    ErrInvalidUTF8          ErrorCode = "invalid UTF-8"  
    ErrMissingBracket       ErrorCode = "missing closing ]"  
    ErrMissingParen         ErrorCode = "missing closing )"  
    ErrMissingRepeatArgument ErrorCode = "missing argument to repeti  
    ErrTrailingBackslash    ErrorCode = "trailing backslash at end  
)
```

func (ErrorCode) [String](#)

```
func (e ErrorCode) String() string
```

type Flags

type Flags uint16

Flags control the behavior of the parser and record information about regexp context.

```
const (
    FoldCase      Flags = 1 << iota // case-insensitive match
    Literal       // treat pattern as literal string
    ClassNL       // allow character classes like
    DotNL         // allow . to match newline
    OneLine       // treat ^ and $ as only matching
    NonGreedy     // make repetition operators default
    PerlX         // allow Perl extensions
    UnicodeGroups // allow \p{Han}, \P{Han} for Unicode
    WasDollar     // regexp OpEndText was $, not \
    Simple        // regexp contains no counted repeats

    MatchNL = ClassNL | DotNL

    Perl      = ClassNL | OneLine | PerlX | UnicodeGroups // as class
    POSIX Flags = 0 // POSIX
)
```

type [Inst](#)

```
type Inst struct {
    Op    InstOp
    Out   uint32 // all but InstMatch, InstFail
    Arg   uint32 // InstAlt, InstAltMatch, InstCapture, InstEmptyWidt
    Rune  []rune
}
```

An Inst is a single instruction in a regular expression program.

func (*Inst) [MatchEmptyWidth](#)

```
func (i *Inst) MatchEmptyWidth(before rune, after rune) bool
```

MatchEmptyWidth returns true if the instruction matches an empty string between the runes before and after. It should only be called when `i.Op == InstEmptyWidth`.

func (*Inst) [MatchRune](#)

```
func (i *Inst) MatchRune(r rune) bool
```

MatchRune returns true if the instruction matches (and consumes) `r`. It should only be called when `i.Op == InstRune`.

func (*Inst) [String](#)

```
func (i *Inst) String() string
```

type [InstOp](#)

```
type InstOp uint8
```

An InstOp is an instruction opcode.

```
const (  
    InstAlt InstOp = iota  
    InstAltMatch  
    InstCapture  
    InstEmptyWidth  
    InstMatch  
    InstFail  
    InstNop  
    InstRune  
    InstRune1  
    InstRuneAny  
    InstRuneAnyNotNL  
)
```

type [Op](#)

```
type Op uint8
```

An Op is a single regular expression operator.

```
const (
    OpNoMatch          Op = 1 + iota // matches no strings
    OpEmptyMatch      // matches empty string
    OpLiteral          // matches Runes sequence
    OpCharClass        // matches Runes interpreted as r
    OpAnyCharNotNL    // matches any character
    OpAnyChar          // matches any character
    OpBeginLine        // matches empty string at beginn
    OpEndLine          // matches empty string at end of
    OpBeginText        // matches empty string at beginn
    OpEndText          // matches empty string at end of
    OpWordBoundary     // matches word boundary `b`
    OpNoWordBoundary  // matches word non-boundary `B`
    OpCapture          // capturing subexpression with i
    OpStar             // matches Sub[0] zero or more ti
    OpPlus            // matches Sub[0] one or more tir
    OpQuest           // matches Sub[0] zero or one tir
    OpRepeat          // matches Sub[0] at least Min ti
    OpConcat          // matches concatenation of Subs
    OpAlternate       // matches alternation of Subs
)
```

type [Prog](#)

```
type Prog struct {
    Inst    []Inst
    Start  int // index of start instruction
    NumCap int // number of InstCapture insts in re
}
```

A Prog is a compiled regular expression program.

func [Compile](#)

```
func Compile(re *Regexp) (*Prog, error)
```

Compile compiles the regexp into a program to be executed. The regexp should have been simplified already (returned from re.Simplify).

func (*Prog) [Prefix](#)

```
func (p *Prog) Prefix() (prefix string, complete bool)
```

Prefix returns a literal string that all matches for the regexp must start with. Complete is true if the prefix is the entire match.

func (*Prog) [StartCond](#)

```
func (p *Prog) StartCond() EmptyOp
```

StartCond returns the leading empty-width conditions that must be true in any match. It returns ^EmptyOp(0) if no matches are possible.

func (*Prog) [String](#)

```
func (p *Prog) String() string
```

type [Regexp](#)

```
type Regexp struct {
    Op      Op // operator
    Flags   Flags
    Sub     []*Regexp // subexpressions, if any
    Sub0    [1]*Regexp // storage for short Sub
    Rune    []rune    // matched runes, for OpLiteral, OpCharClass
    Rune0   [2]rune   // storage for short Rune
    Min, Max int       // min, max for OpRepeat
    Cap     int       // capturing index, for OpCapture
    Name    string    // capturing name, for OpCapture
}
```

A Regexp is a node in a regular expression syntax tree.

func [Parse](#)

```
func Parse(s string, flags Flags) (*Regexp, error)
```

Parse parses a regular expression string *s*, controlled by the specified *Flags*, and returns a regular expression parse tree. The syntax is described in the top-level comment for package `regexp`.

func (*Regexp) [CapNames](#)

```
func (re *Regexp) CapNames() []string
```

CapNames walks the regexp to find the names of capturing groups.

func (*Regexp) [Equal](#)

```
func (x *Regexp) Equal(y *Regexp) bool
```

Equal returns true if *x* and *y* have identical structure.

func (*Regexp) [MaxCap](#)

```
func (re *Regexp) MaxCap() int
```

MaxCap walks the regexp to find the maximum capture index.

func (*Regex) [Simplify](#)

```
func (re *Regex) Simplify() *Regex
```

Simplify returns a regexp equivalent to re but without counted repetitions and with various other simplifications, such as rewriting `/(?:a+)+/` to `/a+/`. The resulting regexp will execute correctly but its string representation will not produce the same parse tree, because capturing parentheses may have been duplicated or removed. For example, the simplified form for `/(x){1,2}/` is `/(x)(x)?/` but both parentheses capture as `$1`. The returned regexp may share structure with or be the original.

func (*Regex) [String](#)

```
func (re *Regex) String() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package runtime

```
import "runtime"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package runtime contains operations that interact with Go's runtime system, such as functions to control goroutines. It also includes the low-level type information used by the reflect package; see reflect's documentation for the programmable interface to the run-time type system.

Index

[Constants](#)

[Variables](#)

[func Breakpoint\(\)](#)

[func CPUProfile\(\) \[\]byte](#)

[func Caller\(skip int\) \(pc uintptr, file string, line int, ok bool\)](#)

[func Callers\(skip int, pc \[\]uintptr\) int](#)

[func GC\(\)](#)

[func GOMAXPROCS\(n int\) int](#)

[func GOROOT\(\) string](#)

[func Goexit\(\)](#)

[func GoroutineProfile\(p \[\]StackRecord\) \(n int, ok bool\)](#)

[func Gosched\(\)](#)

[func LockOSThread\(\)](#)

[func MemProfile\(p \[\]MemProfileRecord, inuseZero bool\) \(n int, ok bool\)](#)

[func NumCPU\(\) int](#)

[func NumCgoCall\(\) int64](#)

[func NumGoroutine\(\) int](#)

[func ReadMemStats\(m *MemStats\)](#)

[func SetCPUProfileRate\(hz int\)](#)

[func SetFinalizer\(x, f interface{}\)](#)

[func Stack\(buf \[\]byte, all bool\) int](#)

[func ThreadCreateProfile\(p \[\]StackRecord\) \(n int, ok bool\)](#)

[func UnlockOSThread\(\)](#)

[func Version\(\) string](#)

[type Error](#)

[type Func](#)

[func FuncForPC\(pc uintptr\) *Func](#)

[func \(f *Func\) Entry\(\) uintptr](#)

[func \(f *Func\) FileLine\(pc uintptr\) \(file string, line int\)](#)

[func \(f *Func\) Name\(\) string](#)

[type MemProfileRecord](#)

[func \(r *MemProfileRecord\) InUseBytes\(\) int64](#)

[func \(r *MemProfileRecord\) InUseObjects\(\) int64](#)

[func \(r *MemProfileRecord\) Stack\(\) \[\]uintptr](#)

[type MemStats](#)

```
type StackRecord  
func (r *StackRecord) Stack() []uintptr  
type TypeAssertionError  
func (e *TypeAssertionError) Error() string  
func (*TypeAssertionError) RuntimeError()
```

Package files

[compiler.go](#) [debug.go](#) [error.go](#) [extern.go](#) [mem.go](#) [softfloat64.go](#) [type.go](#) [zgoarch_amd64.go](#) [zgoos_linux.go](#)
[zruntime_defs_linux_amd64.go](#) [zversion.go](#)

Constants

```
const Compiler = "gc"
```

Compiler is the name of the compiler toolchain that built the running binary.
Known toolchains are:

```
gc      The 5g/6g/8g compiler suite at code.google.com/p/go.  
gccgo   The gccgo front end, part of the GCC compiler suite.
```

```
const GOARCH string = theGoarch
```

GOARCH is the running program's architecture target: 386, amd64, or arm.

```
const GOOS string = theGoos
```

GOOS is the running program's operating system target: one of darwin, freebsd, linux, and so on.

Variables

```
var MemProfileRate int = 512 * 1024
```

MemProfileRate controls the fraction of memory allocations that are recorded and reported in the memory profile. The profiler aims to sample an average of one allocation per MemProfileRate bytes allocated.

To include every allocated block in the profile, set MemProfileRate to 1. To turn off profiling entirely, set MemProfileRate to 0.

The tools that process the memory profiles assume that the profile rate is constant across the lifetime of the program and equal to the current value. Programs that change the memory profiling rate should do so just once, as early as possible in the execution of the program (for example, at the beginning of main).

func Breakpoint

func Breakpoint()

Breakpoint() executes a breakpoint trap.

func [CPUProfile](#)

```
func CPUProfile() []byte
```

CPUProfile returns the next chunk of binary CPU profiling stack trace data, blocking until data is available. If profiling is turned off and all the profile data accumulated while it was on has been returned, CPUProfile returns nil. The caller must save the returned data before calling CPUProfile again. Most clients should use the runtime/pprof package or the testing package's `-test.cpuprofile` flag instead of calling CPUProfile directly.

func [Caller](#)

```
func Caller(skip int) (pc uintptr, file string, line int, ok bool)
```

Caller reports file and line number information about function invocations on the calling goroutine's stack. The argument skip is the number of stack frames to ascend, with 1 identifying the caller of Caller. (For historical reasons the meaning of skip differs between Caller and Callers.) The return values report the program counter, file name, and line number within the file of the corresponding call. The boolean ok is false if it was not possible to recover the information.

func [Callers](#)

```
func Callers(skip int, pc []uintptr) int
```

Callers fills the slice pc with the program counters of function invocations on the calling goroutine's stack. The argument skip is the number of stack frames to skip before recording in pc, with 0 starting at the caller of Callers. It returns the number of entries written to pc.

func [GC](#)

func GC()

GC runs a garbage collection.

func GOMAXPROCS

```
func GOMAXPROCS(n int) int
```

GOMAXPROCS sets the maximum number of CPUs that can be executing simultaneously and returns the previous setting. If $n < 1$, it does not change the current setting. The number of logical CPUs on the local machine can be queried with NumCPU. This call will go away when the scheduler improves.

func GOROOT

```
func GOROOT() string
```

GOROOT returns the root of the Go tree. It uses the GOROOT environment variable, if set, or else the root used during the Go build.

func [Goexit](#)

```
func Goexit()
```

Goexit terminates the goroutine that calls it. No other goroutine is affected. Goexit runs all deferred calls before terminating the goroutine.

func [GoroutineProfile](#)

```
func GoroutineProfile(p []StackRecord) (n int, ok bool)
```

GoroutineProfile returns n, the number of records in the active goroutine stack profile. If $\text{len}(p) \geq n$, GoroutineProfile copies the profile into p and returns n, true. If $\text{len}(p) < n$, GoroutineProfile does not change p and returns n, false.

Most clients should use the runtime/pprof package instead of calling GoroutineProfile directly.

func [Gosched](#)

```
func Gosched()
```

Gosched yields the processor, allowing other goroutines to run. It does not suspend the current goroutine, so execution resumes automatically.

func LockOSThread

```
func LockOSThread()
```

LockOSThread wires the calling goroutine to its current operating system thread. Until the calling goroutine exits or calls `UnlockOSThread`, it will always execute in that thread, and no other goroutine can.

func [MemProfile](#)

```
func MemProfile(p []MemProfileRecord, inuseZero bool) (n int, ok bool)
```

MemProfile returns n, the number of records in the current memory profile. If len(p) >= n, MemProfile copies the profile into p and returns n, true. If len(p) < n, MemProfile does not change p and returns n, false.

If inuseZero is true, the profile includes allocation records where r.AllocBytes > 0 but r.AllocBytes == r.FreeBytes. These are sites where memory was allocated, but it has all been released back to the runtime.

Most clients should use the runtime/pprof package or the testing package's -test.memprofile flag instead of calling MemProfile directly.

func [NumCPU](#)

```
func NumCPU() int
```

NumCPU returns the number of logical CPUs on the local machine.

func [NumCgoCall](#)

```
func NumCgoCall() int64
```

NumCgoCall returns the number of cgo calls made by the current process.

func [NumGoroutine](#)

```
func NumGoroutine() int
```

NumGoroutine returns the number of goroutines that currently exist.

func [ReadMemStats](#)

```
func ReadMemStats(m *MemStats)
```

ReadMemStats populates m with memory allocator statistics.

func [SetCPUProfileRate](#)

```
func SetCPUProfileRate(hz int)
```

SetCPUProfileRate sets the CPU profiling rate to hz samples per second. If hz ≤ 0 , SetCPUProfileRate turns off profiling. If the profiler is on, the rate cannot be changed without first turning it off. Most clients should use the runtime/pprof package or the testing package's -test.cpuprofile flag instead of calling SetCPUProfileRate directly.

func [SetFinalizer](#)

```
func SetFinalizer(x, f interface{})
```

SetFinalizer sets the finalizer associated with `x` to `f`. When the garbage collector finds an unreachable block with an associated finalizer, it clears the association and runs `f(x)` in a separate goroutine. This makes `x` reachable again, but now without an associated finalizer. Assuming that SetFinalizer is not called again, the next time the garbage collector sees that `x` is unreachable, it will free `x`.

SetFinalizer(`x`, nil) clears any finalizer associated with `x`.

The argument `x` must be a pointer to an object allocated by calling `new` or by taking the address of a composite literal. The argument `f` must be a function that takes a single argument of `x`'s type and can have arbitrary ignored return values. If either of these is not true, SetFinalizer aborts the program.

Finalizers are run in dependency order: if `A` points at `B`, both have finalizers, and they are otherwise unreachable, only the finalizer for `A` runs; once `A` is freed, the finalizer for `B` can run. If a cyclic structure includes a block with a finalizer, that cycle is not guaranteed to be garbage collected and the finalizer is not guaranteed to run, because there is no ordering that respects the dependencies.

The finalizer for `x` is scheduled to run at some arbitrary time after `x` becomes unreachable. There is no guarantee that finalizers will run before a program exits, so typically they are useful only for releasing non-memory resources associated with an object during a long-running program. For example, an `os.File` object could use a finalizer to close the associated operating system file descriptor when a program discards an `os.File` without calling `Close`, but it would be a mistake to depend on a finalizer to flush an in-memory I/O buffer such as a `bufio.Writer`, because the buffer would not be flushed at program exit.

A single goroutine runs all finalizers for a program, sequentially. If a finalizer must run for a long time, it should do so by starting a new goroutine.

func [Stack](#)

```
func Stack(buf []byte, all bool) int
```

Stack formats a stack trace of the calling goroutine into buf and returns the number of bytes written to buf. If all is true, Stack formats stack traces of all other goroutines into buf after the trace for the current goroutine.

func [ThreadCreateProfile](#)

```
func ThreadCreateProfile(p []StackRecord) (n int, ok bool)
```

ThreadCreateProfile returns n, the number of records in the thread creation profile. If $\text{len}(p) \geq n$, ThreadCreateProfile copies the profile into p and returns n, true. If $\text{len}(p) < n$, ThreadCreateProfile does not change p and returns n, false.

Most clients should use the runtime/pprof package instead of calling ThreadCreateProfile directly.

func [UnlockOSThread](#)

```
func UnlockOSThread()
```

UnlockOSThread unwires the calling goroutine from its fixed operating system thread. If the calling goroutine has not called LockOSThread, UnlockOSThread is a no-op.

func [Version](#)

```
func Version() string
```

Version returns the Go tree's version string. It is either a sequence number or, when possible, a release tag like "release.2010-03-04". A trailing + indicates that the tree had local modifications at the time of the build.

type [Error](#)

```
type Error interface {
    error

    // RuntimeError is a no-op function but
    // serves to distinguish types that are runtime
    // errors from ordinary errors: a type is a
    // runtime error if it has a RuntimeError method.
    RuntimeError()
}
```

The Error interface identifies a run time error.

type [Func](#)

```
type Func struct {  
    // contains filtered or unexported fields  
}
```

func [FuncForPC](#)

```
func FuncForPC(pc uintptr) *Func
```

FuncForPC returns a *Func describing the function that contains the given program counter address, or else nil.

func (*Func) [Entry](#)

```
func (f *Func) Entry() uintptr
```

Entry returns the entry address of the function.

func (*Func) [FileLine](#)

```
func (f *Func) FileLine(pc uintptr) (file string, line int)
```

FileLine returns the file name and line number of the source code corresponding to the program counter pc. The result will not be accurate if pc is not a program counter within f.

func (*Func) [Name](#)

```
func (f *Func) Name() string
```

Name returns the name of the function.

type [MemProfileRecord](#)

```
type MemProfileRecord struct {
    AllocBytes, FreeBytes    int64        // number of bytes allocat
    AllocObjects, FreeObjects int64        // number of objects alloc
    Stack0                  [32]uintptr // stack trace for this re
}
```

A MemProfileRecord describes the live objects allocated by a particular call sequence (stack trace).

func (*MemProfileRecord) [InUseBytes](#)

```
func (r *MemProfileRecord) InUseBytes() int64
```

InUseBytes returns the number of bytes in use (AllocBytes - FreeBytes).

func (*MemProfileRecord) [InUseObjects](#)

```
func (r *MemProfileRecord) InUseObjects() int64
```

InUseObjects returns the number of objects in use (AllocObjects - FreeObjects).

func (*MemProfileRecord) [Stack](#)

```
func (r *MemProfileRecord) Stack() []uintptr
```

Stack returns the stack trace associated with the record, a prefix of r.Stack0.

type [MemStats](#)

```
type MemStats struct {
    // General statistics.
    Alloc      uint64 // bytes allocated and still in use
    TotalAlloc uint64 // bytes allocated (even if freed)
    Sys        uint64 // bytes obtained from system (should be sum o
    Lookups    uint64 // number of pointer lookups
    Mallocs    uint64 // number of mallocs
    Frees      uint64

    // Main allocation heap statistics.
    HeapAlloc   uint64 // bytes allocated and still in use
    HeapSys     uint64 // bytes obtained from system
    HeapIdle    uint64 // bytes in idle spans
    HeapInuse   uint64 // bytes in non-idle span
    HeapReleased uint64 // bytes released to the OS
    HeapObjects uint64

    // Low-level fixed-size structure allocator statistics.
    // Inuse is bytes used now.
    // Sys is bytes obtained from system.
    StackInuse uint64 // bootstrap stacks
    StackSys   uint64
    MSpanInuse uint64 // mspan structures
    MSpanSys   uint64
    MCacheInuse uint64 // mcache structures
    MCacheSys   uint64
    BuckHashSys uint64

    // Garbage collector statistics.
    NextGC      uint64 // next run in HeapAlloc time (bytes)
    LastGC      uint64 // last run in absolute time (ns)
    PauseTotalNs uint64
    PauseNs     [256]uint64 // most recent GC pause times
    NumGC       uint32
    EnableGC    bool
    DebugGC     bool

    // Per-size allocation statistics.
    // 61 is NumSizeClasses in the C code.
    BySize [61]struct {
        Size      uint32
        Mallocs   uint64
        Frees     uint64
    }
}
```

A MemStats records statistics about the memory allocator.

type [StackRecord](#)

```
type StackRecord struct {  
    Stack0 [32]uintptr // stack trace for this record; ends at first  
}
```

A StackRecord describes a single execution stack.

func (*StackRecord) [Stack](#)

```
func (r *StackRecord) Stack() []uintptr
```

Stack returns the stack trace associated with the record, a prefix of r.Stack0.

type [TypeAssertionError](#)

```
type TypeAssertionError struct {  
    // contains filtered or unexported fields  
}
```

A TypeAssertionError explains a failed type assertion.

func (*TypeAssertionError) [Error](#)

```
func (e *TypeAssertionError) Error() string
```

func (*TypeAssertionError) [RuntimeError](#)

```
func (*TypeAssertionError) RuntimeError()
```

Subdirectories

Name	Synopsis
------	----------

cgo	Package cgo contains runtime support for code generated by the cgo tool.
---------------------	--

debug	Package debug contains facilities for programs to debug themselves while they are running.
-----------------------	--

pprof	Package pprof writes runtime profiling data in the format expected by the pprof visualization tool.
-----------------------	---

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package cgo

```
import "runtime/cgo"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `cgo` contains runtime support for code generated by the `cgo` tool. See the documentation for the `cgo` command for details on using `cgo`.

Index

Package files

cgo.go

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package debug

```
import "runtime/debug"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package debug contains facilities for programs to debug themselves while they are running.

Index

[func PrintStack\(\)](#)
[func Stack\(\) \[\]byte](#)

Package files

[stack.go](#)

func [PrintStack](#)

```
func PrintStack()
```

PrintStack prints to standard error the stack trace returned by Stack.

func [Stack](#)

```
func Stack() []byte
```

Stack returns a formatted stack trace of the goroutine that calls it. For each routine, it includes the source line information and PC value, then attempts to discover, for Go functions, the calling function or method and the text of the line containing the invocation.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package pprof

```
import "runtime/pprof"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package pprof writes runtime profiling data in the format expected by the pprof visualization tool. For more information about pprof, see <http://code.google.com/p/google-perftools/>.

Index

[func Profiles\(\) \[\]*Profile](#)

[func StartCPUProfile\(w io.Writer\) error](#)

[func StopCPUProfile\(\)](#)

[func WriteHeapProfile\(w io.Writer\) error](#)

[type Profile](#)

[func Lookup\(name string\) *Profile](#)

[func NewProfile\(name string\) *Profile](#)

[func \(p *Profile\) Add\(value interface{}, skip int\)](#)

[func \(p *Profile\) Count\(\) int](#)

[func \(p *Profile\) Name\(\) string](#)

[func \(p *Profile\) Remove\(value interface{}\)](#)

[func \(p *Profile\) WriteTo\(w io.Writer, debug int\) error](#)

[Bugs](#)

Package files

[pprof.go](#)

func [Profiles](#)

```
func Profiles() []*Profile
```

Profiles returns a slice of all the known profiles, sorted by name.

func [StartCPUProfile](#)

```
func StartCPUProfile(w io.Writer) error
```

StartCPUProfile enables CPU profiling for the current process. While profiling, the profile will be buffered and written to w. StartCPUProfile returns an error if profiling is already enabled.

func [StopCPUProfile](#)

```
func StopCPUProfile()
```

StopCPUProfile stops the current CPU profile, if any. StopCPUProfile only returns after all the writes for the profile have completed.

func [WriteHeapProfile](#)

```
func WriteHeapProfile(w io.Writer) error
```

WriteHeapProfile is shorthand for `Lookup("heap").WriteTo(w, 0)`. It is preserved for backwards compatibility.

type [Profile](#)

```
type Profile struct {  
    // contains filtered or unexported fields  
}
```

A Profile is a collection of stack traces showing the call sequences that led to instances of a particular event, such as allocation. Packages can create and maintain their own profiles; the most common use is for tracking resources that must be explicitly closed, such as files or network connections.

A Profile's methods can be called from multiple goroutines simultaneously.

Each Profile has a unique name. A few profiles are predefined:

```
goroutine    - stack traces of all current goroutines  
heap         - a sampling of all heap allocations  
threadcreate - stack traces that led to the creation of new OS threa
```

These predefined profiles maintain themselves and panic on an explicit Add or Remove method call.

The CPU profile is not available as a Profile. It has a special API, the StartCPUProfile and StopCPUProfile functions, because it streams output to a writer during profiling.

func [Lookup](#)

```
func Lookup(name string) *Profile
```

Lookup returns the profile with the given name, or nil if no such profile exists.

func [NewProfile](#)

```
func NewProfile(name string) *Profile
```

NewProfile creates a new profile with the given name. If a profile with that name already exists, NewProfile panics. The convention is to use a 'import/path.' prefix to create separate name spaces for each package.

func (*Profile) [Add](#)

```
func (p *Profile) Add(value interface{}, skip int)
```

Add adds the current execution stack to the profile, associated with value. Add stores value in an internal map, so value must be suitable for use as a map key and will not be garbage collected until the corresponding call to Remove. Add panics if the profile already contains a stack for value.

The skip parameter has the same meaning as runtime.Caller's skip and controls where the stack trace begins. Passing skip=0 begins the trace in the function calling Add. For example, given this execution stack:

```
Add  
called from rpc.NewClient  
called from mypkg.Run  
called from main.main
```

Passing skip=0 begins the stack trace at the call to Add inside rpc.NewClient. Passing skip=1 begins the stack trace at the call to NewClient inside mypkg.Run.

func (*Profile) [Count](#)

```
func (p *Profile) Count() int
```

Count returns the number of execution stacks currently in the profile.

func (*Profile) [Name](#)

```
func (p *Profile) Name() string
```

Name returns this profile's name, which can be passed to Lookup to reobtain the profile.

func (*Profile) [Remove](#)

```
func (p *Profile) Remove(value interface{})
```

Remove removes the execution stack associated with value from the profile. It is a no-op if the value is not in the profile.

func (*Profile) [WriteTo](#)

```
func (p *Profile) WriteTo(w io.Writer, debug int) error
```

WriteTo writes a pprof-formatted snapshot of the profile to w. If a write to w returns an error, WriteTo returns that error. Otherwise, WriteTo returns nil.

The debug parameter enables additional output. Passing debug=0 prints only the hexadecimal addresses that pprof needs. Passing debug=1 adds comments translating addresses to function names and line numbers, so that a programmer can read the profile without tools.

The predefined profiles may assign meaning to other debug values; for example, when printing the "goroutine" profile, debug=2 means to print the goroutine stacks in the same form that a Go program uses when dying due to an unrecovered panic.

Bugs

A bug in the OS X Snow Leopard 64-bit kernel prevents CPU profiling from giving accurate results on that system.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package sort

```
import "sort"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package sort provides primitives for sorting slices and user-defined collections.

Index

[func Float64s\(a \[\]float64\)](#)
[func Float64sAreSorted\(a \[\]float64\) bool](#)
[func Ints\(a \[\]int\)](#)
[func IntsAreSorted\(a \[\]int\) bool](#)
[func IsSorted\(data Interface\) bool](#)
[func Search\(n int, f func\(int\) bool\) int](#)
[func SearchFloat64s\(a \[\]float64, x float64\) int](#)
[func SearchInts\(a \[\]int, x int\) int](#)
[func SearchStrings\(a \[\]string, x string\) int](#)
[func Sort\(data Interface\)](#)
[func Strings\(a \[\]string\)](#)
[func StringsAreSorted\(a \[\]string\) bool](#)
[type Float64Slice](#)
 [func \(p Float64Slice\) Len\(\) int](#)
 [func \(p Float64Slice\) Less\(i, j int\) bool](#)
 [func \(p Float64Slice\) Search\(x float64\) int](#)
 [func \(p Float64Slice\) Sort\(\)](#)
 [func \(p Float64Slice\) Swap\(i, j int\)](#)
[type IntSlice](#)
 [func \(p IntSlice\) Len\(\) int](#)
 [func \(p IntSlice\) Less\(i, j int\) bool](#)
 [func \(p IntSlice\) Search\(x int\) int](#)
 [func \(p IntSlice\) Sort\(\)](#)
 [func \(p IntSlice\) Swap\(i, j int\)](#)
[type Interface](#)
[type StringSlice](#)
 [func \(p StringSlice\) Len\(\) int](#)
 [func \(p StringSlice\) Less\(i, j int\) bool](#)
 [func \(p StringSlice\) Search\(x string\) int](#)
 [func \(p StringSlice\) Sort\(\)](#)
 [func \(p StringSlice\) Swap\(i, j int\)](#)

Examples

[Interface](#)

[Interface \(Reverse\)](#)
[Ints](#)

Package files

[search.go](#) [sort.go](#)

func [Float64s](#)

```
func Float64s(a []float64)
```

Float64s sorts a slice of float64s in increasing order.

func [Float64sAreSorted](#)

```
func Float64sAreSorted(a []float64) bool
```

Float64sAreSorted tests whether a slice of float64s is sorted in increasing order.

func [Ints](#)

```
func Ints(a []int)
```

Ints sorts a slice of ints in increasing order.

? Example

? Example

Code:

```
s := []int{5, 2, 6, 3, 1, 4} // unsorted
sort.Ints(s)
fmt.Println(s)
```

Output:

```
[1 2 3 4 5 6]
```

func IntsAreSorted

```
func IntsAreSorted(a []int) bool
```

IntsAreSorted tests whether a slice of ints is sorted in increasing order.

func IsSorted

```
func IsSorted(data Interface) bool
```

IsSorted reports whether data is sorted.

func [Search](#)

```
func Search(n int, f func(int) bool) int
```

Search uses binary search to find and return the smallest index i in $[0, n)$ at which $f(i)$ is true, assuming that on the range $[0, n)$, $f(i) == \text{true}$ implies $f(i+1) == \text{true}$. That is, Search requires that f is false for some (possibly empty) prefix of the input range $[0, n)$ and then true for the (possibly empty) remainder; Search returns the first true index. If there is no such index, Search returns n . Search calls $f(i)$ only for i in the range $[0, n)$.

A common use of Search is to find the index i for a value x in a sorted, indexable data structure such as an array or slice. In this case, the argument f , typically a closure, captures the value to be searched for, and how the data structure is indexed and ordered.

For instance, given a slice `data` sorted in ascending order, the call `Search(len(data), func(i int) bool { return data[i] >= 23 })` returns the smallest index i such that `data[i] >= 23`. If the caller wants to find whether 23 is in the slice, it must test `data[i] == 23` separately.

Searching data sorted in descending order would use the `<=` operator instead of the `>=` operator.

To complete the example above, the following code tries to find the value x in an integer slice `data` sorted in ascending order:

```
x := 23
i := sort.Search(len(data), func(i int) bool { return data[i] >= x })
if i < len(data) && data[i] == x {
    // x is present at data[i]
} else {
    // x is not present in data,
    // but i is the index where it would be inserted.
}
```

As a more whimsical example, this program guesses your number:

```
func GuessingGame() {
    var s string
    fmt.Printf("Pick an integer from 0 to 100.\n")
}
```

```
    answer := sort.Search(100, func(i int) bool {
        fmt.Printf("Is your number <= %d? ", i)
        fmt.Scanf("%s", &s)
        return s != "" && s[0] == 'y'
    })
    fmt.Printf("Your number is %d.\n", answer)
}
```

func [SearchFloat64s](#)

```
func SearchFloat64s(a []float64, x float64) int
```

SearchFloat64s searches for x in a sorted slice of float64s and returns the index as specified by Search. The slice must be sorted in ascending order.

func [SearchInts](#)

```
func SearchInts(a []int, x int) int
```

SearchInts searches for x in a sorted slice of ints and returns the index as specified by Search. The slice must be sorted in ascending order.

func [SearchStrings](#)

```
func SearchStrings(a []string, x string) int
```

SearchStrings searches for x slice a sorted slice of strings and returns the index as specified by Search. The slice must be sorted in ascending order.

func [Sort](#)

```
func Sort(data Interface)
```

Sort sorts data. It makes one call to data.Len to determine n, and $O(n \cdot \log(n))$ calls to data.Less and data.Swap. The sort is not guaranteed to be stable.

func Strings

```
func Strings(a []string)
```

Strings sorts a slice of strings in increasing order.

func StringsAreSorted

```
func StringsAreSorted(a []string) bool
```

StringsAreSorted tests whether a slice of strings is sorted in increasing order.

type [Float64Slice](#)

```
type Float64Slice []float64
```

Float64Slice attaches the methods of Interface to []float64, sorting in increasing order.

func (Float64Slice) [Len](#)

```
func (p Float64Slice) Len() int
```

func (Float64Slice) [Less](#)

```
func (p Float64Slice) Less(i, j int) bool
```

func (Float64Slice) [Search](#)

```
func (p Float64Slice) Search(x float64) int
```

Search returns the result of applying SearchFloat64s to the receiver and x.

func (Float64Slice) [Sort](#)

```
func (p Float64Slice) Sort()
```

Sort is a convenience method.

func (Float64Slice) [Swap](#)

```
func (p Float64Slice) Swap(i, j int)
```

type [IntSlice](#)

```
type IntSlice []int
```

IntSlice attaches the methods of Interface to []int, sorting in increasing order.

func (IntSlice) [Len](#)

```
func (p IntSlice) Len() int
```

func (IntSlice) [Less](#)

```
func (p IntSlice) Less(i, j int) bool
```

func (IntSlice) [Search](#)

```
func (p IntSlice) Search(x int) int
```

Search returns the result of applying SearchInts to the receiver and x.

func (IntSlice) [Sort](#)

```
func (p IntSlice) Sort()
```

Sort is a convenience method.

func (IntSlice) [Swap](#)

```
func (p IntSlice) Swap(i, j int)
```

type [Interface](#)

```
type Interface interface {
    // Len is the number of elements in the collection.
    Len() int
    // Less returns whether the element with index i should sort
    // before the element with index j.
    Less(i, j int) bool
    // Swap swaps the elements with indexes i and j.
    Swap(i, j int)
}
```

A type, typically a collection, that satisfies `sort.Interface` can be sorted by the routines in this package. The methods require that the elements of the collection be enumerated by an integer index.

? Example

? Example

Code:

```
package sort_test

import (
    "fmt"
    "sort"
)

type Grams int

func (g Grams) String() string { return fmt.Sprintf("%dg", int(g)) }

type Organ struct {
    Name    string
    Weight  Grams
}

type Organs []*Organ

func (s Organs) Len() int      { return len(s) }
func (s Organs) Swap(i, j int) { s[i], s[j] = s[j], s[i] }

// ByName implements sort.Interface by providing Less and using the
// Swap methods of the embedded Organs value.
```

```

type ByName struct{ Organs }

func (s ByName) Less(i, j int) bool { return s.Organs[i].Name < s.Or

// ByWeight implements sort.Interface by providing Less and using th
// Swap methods of the embedded Organs value.
type ByWeight struct{ Organs }

func (s ByWeight) Less(i, j int) bool { return s.Organs[i].Weight <

func ExampleInterface() {
    s := []*Organ{
        {"brain", 1340},
        {"heart", 290},
        {"liver", 1494},
        {"pancreas", 131},
        {"prostate", 62},
        {"spleen", 162},
    }

    sort.Sort(ByWeight{s})
    fmt.Println("Organs by weight:")
    printOrgans(s)

    sort.Sort(ByName{s})
    fmt.Println("Organs by name:")
    printOrgans(s)

    // Output:
    // Organs by weight:
    // prostate (62g)
    // pancreas (131g)
    // spleen (162g)
    // heart (290g)
    // brain (1340g)
    // liver (1494g)
    // Organs by name:
    // brain (1340g)
    // heart (290g)
    // liver (1494g)
    // pancreas (131g)
    // prostate (62g)
    // spleen (162g)
}

func printOrgans(s []*Organ) {
    for _, o := range s {
        fmt.Printf("%-8s (%v)\n", o.Name, o.Weight)
    }
}

```

```
}
```

? Example (Reverse)

? Example (Reverse)

Code:

```
package sort_test

import (
    "fmt"
    "sort"
)

// Reverse embeds a sort.Interface value and implements a reverse so
// that value.
type Reverse struct {
    // This embedded Interface permits Reverse to use the methods of
    // another Interface implementation.
    sort.Interface
}

// Less returns the opposite of the embedded implementation's Less m
func (r Reverse) Less(i, j int) bool {
    return r.Interface.Less(j, i)
}

func ExampleInterface_reverse() {
    s := []int{5, 2, 6, 3, 1, 4} // unsorted
    sort.Sort(Reverse{sort.IntSlice(s)})
    fmt.Println(s)
    // Output: [6 5 4 3 2 1]
}
```

type [StringSlice](#)

```
type StringSlice []string
```

StringSlice attaches the methods of Interface to []string, sorting in increasing order.

func (StringSlice) [Len](#)

```
func (p StringSlice) Len() int
```

func (StringSlice) [Less](#)

```
func (p StringSlice) Less(i, j int) bool
```

func (StringSlice) [Search](#)

```
func (p StringSlice) Search(x string) int
```

Search returns the result of applying SearchStrings to the receiver and x.

func (StringSlice) [Sort](#)

```
func (p StringSlice) Sort()
```

Sort is a convenience method.

func (StringSlice) [Swap](#)

```
func (p StringSlice) Swap(i, j int)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package strconv

```
import "strconv"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package strconv implements conversions to and from string representations of basic data types.

Index

Constants

Variables

[func AppendBool\(dst \[\]byte, b bool\) \[\]byte](#)

[func AppendFloat\(dst \[\]byte, f float64, fmt byte, prec int, bitSize int\) \[\]byte](#)

[func AppendInt\(dst \[\]byte, i int64, base int\) \[\]byte](#)

[func AppendQuote\(dst \[\]byte, s string\) \[\]byte](#)

[func AppendQuoteRune\(dst \[\]byte, r rune\) \[\]byte](#)

[func AppendQuoteRuneToASCII\(dst \[\]byte, r rune\) \[\]byte](#)

[func AppendQuoteToASCII\(dst \[\]byte, s string\) \[\]byte](#)

[func AppendUint\(dst \[\]byte, i uint64, base int\) \[\]byte](#)

[func Atoi\(s string\) \(i int, err error\)](#)

[func CanBackquote\(s string\) bool](#)

[func FormatBool\(b bool\) string](#)

[func FormatFloat\(f float64, fmt byte, prec, bitSize int\) string](#)

[func FormatInt\(i int64, base int\) string](#)

[func FormatUint\(i uint64, base int\) string](#)

[func IsPrint\(r rune\) bool](#)

[func Itoa\(i int\) string](#)

[func ParseBool\(str string\) \(value bool, err error\)](#)

[func ParseFloat\(s string, bitSize int\) \(f float64, err error\)](#)

[func ParseInt\(s string, base int, bitSize int\) \(i int64, err error\)](#)

[func ParseUint\(s string, b int, bitSize int\) \(n uint64, err error\)](#)

[func Quote\(s string\) string](#)

[func QuoteRune\(r rune\) string](#)

[func QuoteRuneToASCII\(r rune\) string](#)

[func QuoteToASCII\(s string\) string](#)

[func Unquote\(s string\) \(t string, err error\)](#)

[func UnquoteChar\(s string, quote byte\) \(value rune, multibyte bool, tail string, err error\)](#)

[type NumError](#)

[func \(e *NumError\) Error\(\) string](#)

Package files

[atob.go](#) [atof.go](#) [atoi.go](#) [decimal.go](#) [extfloat.go](#) [ftoa.go](#) [isprint.go](#) [itoa.go](#) [quote.go](#)

Constants

```
const IntSize = intSize // number of bits in int, uint (32 or 64)
```

Variables

```
var ErrRange = errors.New("value out of range")
```

ErrRange indicates that a value is out of range for the target type.

```
var ErrSyntax = errors.New("invalid syntax")
```

ErrSyntax indicates that a value does not have the right syntax for the target type.

func AppendBool

```
func AppendBool(dst []byte, b bool) []byte
```

AppendBool appends "true" or "false", according to the value of b, to dst and returns the extended buffer.

func [AppendFloat](#)

```
func AppendFloat(dst []byte, f float64, fmt byte, prec int, bitSize
```

AppendFloat appends the string form of the floating-point number `f`, as generated by `FormatFloat`, to `dst` and returns the extended buffer.

func [AppendInt](#)

```
func AppendInt(dst []byte, i int64, base int) []byte
```

AppendInt appends the string form of the integer *i*, as generated by `FormatInt`, to *dst* and returns the extended buffer.

func [AppendQuote](#)

```
func AppendQuote(dst []byte, s string) []byte
```

AppendQuote appends a double-quoted Go string literal representing s, as generated by Quote, to dst and returns the extended buffer.

func [AppendQuoteRune](#)

```
func AppendQuoteRune(dst []byte, r rune) []byte
```

AppendQuoteRune appends a single-quoted Go character literal representing the rune, as generated by QuoteRune, to dst and returns the extended buffer.

func [AppendQuoteRuneToASCII](#)

```
func AppendQuoteRuneToASCII(dst []byte, r rune) []byte
```

AppendQuoteRune appends a single-quoted Go character literal representing the rune, as generated by QuoteRuneToASCII, to dst and returns the extended buffer.

func [AppendQuoteToASCII](#)

```
func AppendQuoteToASCII(dst []byte, s string) []byte
```

AppendQuoteToASCII appends a double-quoted Go string literal representing s, as generated by QuoteToASCII, to dst and returns the extended buffer.

func [AppendUint](#)

```
func AppendUint(dst []byte, i uint64, base int) []byte
```

AppendUint appends the string form of the unsigned integer *i*, as generated by `FormatUint`, to *dst* and returns the extended buffer.

func [Atoi](#)

```
func Atoi(s string) (i int, err error)
```

Atoi is shorthand for `ParseInt(s, 10, 0)`.

func [CanBackquote](#)

```
func CanBackquote(s string) bool
```

CanBackquote returns whether the string `s` would be a valid Go string literal if enclosed in backquotes.

func FormatBool

```
func FormatBool(b bool) string
```

FormatBool returns "true" or "false" according to the value of b

func [FormatFloat](#)

```
func FormatFloat(f float64, fmt byte, prec, bitSize int) string
```

FormatFloat converts the floating-point number `f` to a string, according to the format `fmt` and precision `prec`. It rounds the result assuming that the original was obtained from a floating-point value of `bitSize` bits (32 for float32, 64 for float64).

The format `fmt` is one of 'b' (-ddddpddd, a binary exponent), 'e' (-d.ddddedd, a decimal exponent), 'E' (-d.ddddEdd, a decimal exponent), 'f' (-ddd.dddd, no exponent), 'g' ('e' for large exponents, 'f' otherwise), or 'G' ('E' for large exponents, 'f' otherwise).

The precision `prec` controls the number of digits (excluding the exponent) printed by the 'e', 'E', 'f', 'g', and 'G' formats. For 'e', 'E', and 'f' it is the number of digits after the decimal point. For 'g' and 'G' it is the total number of digits. The special precision -1 uses the smallest number of digits necessary such that ParseFloat will return `f` exactly.

func [FormatInt](#)

```
func FormatInt(i int64, base int) string
```

FormatInt returns the string representation of i in the given base.

func [FormatUint](#)

```
func FormatUint(i uint64, base int) string
```

FormatUint returns the string representation of i in the given base.

func [IsPrint](#)

```
func IsPrint(r rune) bool
```

IsPrint reports whether the rune is defined as printable by Go, with the same definition as `unicode.IsPrint`: letters, numbers, punctuation, symbols and ASCII space.

func [Itoa](#)

```
func Itoa(i int) string
```

Itoa is shorthand for `FormatInt(i, 10)`.

func [ParseBool](#)

```
func ParseBool(str string) (value bool, err error)
```

ParseBool returns the boolean value represented by the string. It accepts 1, t, T, TRUE, true, True, 0, f, F, FALSE, false, False. Any other value returns an error.

func [ParseFloat](#)

```
func ParseFloat(s string, bitSize int) (f float64, err error)
```

ParseFloat converts the string `s` to a floating-point number with the precision specified by `bitSize`: 32 for `float32`, or 64 for `float64`. When `bitSize=32`, the result still has type `float64`, but it will be convertible to `float32` without changing its value.

If `s` is well-formed and near a valid floating point number, ParseFloat returns the nearest floating point number rounded using IEEE754 unbiased rounding.

The errors that ParseFloat returns have concrete type `*NumError` and include `err.Num = s`.

If `s` is not syntactically well-formed, ParseFloat returns `err.Error = ErrSyntax`.

If `s` is syntactically well-formed but is more than 1/2 ULP away from the largest floating point number of the given size, ParseFloat returns `f = Inf`, `err.Error = ErrRange`.

func [ParseInt](#)

```
func ParseInt(s string, base int, bitSize int) (i int64, err error)
```

ParseInt interprets a string `s` in the given base (2 to 36) and returns the corresponding value `i`. If `base == 0`, the base is implied by the string's prefix: base 16 for "0x", base 8 for "0", and base 10 otherwise.

The `bitSize` argument specifies the integer type that the result must fit into. Bit sizes 0, 8, 16, 32, and 64 correspond to `int`, `int8`, `int16`, `int32`, and `int64`.

The errors that ParseInt returns have concrete type `*NumError` and include `err.Num = s`. If `s` is empty or contains invalid digits, `err.Error = ErrSyntax`; if the value corresponding to `s` cannot be represented by a signed integer of the given size, `err.Error = ErrRange`.

func [ParseUint](#)

```
func ParseUint(s string, b int, bitSize int) (n uint64, err error)
```

ParseUint is like ParseInt but for unsigned numbers.

func [Quote](#)

```
func Quote(s string) string
```

Quote returns a double-quoted Go string literal representing s. The returned string uses Go escape sequences (`\t`, `\n`, `\xFF`, `\u0100`) for control characters and non-printable characters as defined by `IsPrint`.

func [QuoteRune](#)

```
func QuoteRune(r rune) string
```

QuoteRune returns a single-quoted Go character literal representing the rune. The returned string uses Go escape sequences (`\t`, `\n`, `\xFF`, `\u0100`) for control characters and non-printable characters as defined by `IsPrint`.

func QuoteRuneToASCII

```
func QuoteRuneToASCII(r rune) string
```

QuoteRuneToASCII returns a single-quoted Go character literal representing the rune. The returned string uses Go escape sequences (`\t`, `\n`, `\xFF`, `\u0100`) for non-ASCII characters and non-printable characters as defined by `IsPrint`.

func QuoteToASCII

```
func QuoteToASCII(s string) string
```

QuoteToASCII returns a double-quoted Go string literal representing s. The returned string uses Go escape sequences (`\t`, `\n`, `\xFF`, `\u0100`) for non-ASCII characters and non-printable characters as defined by `IsPrint`.

func [Unquote](#)

```
func Unquote(s string) (t string, err error)
```

Unquote interprets s as a single-quoted, double-quoted, or backquoted Go string literal, returning the string value that s quotes. (If s is single-quoted, it would be a Go character literal; Unquote returns the corresponding one-character string.)

func UnquoteChar

`func UnquoteChar(s string, quote byte) (value rune, multibyte bool,`

`UnquoteChar` decodes the first character or byte in the escaped string or character literal represented by the string `s`. It returns four values:

- 1) `value`, the decoded Unicode code point or byte value;
- 2) `multibyte`, a boolean indicating whether the decoded character requires a multibyte escape;
- 3) `tail`, the remainder of the string after the character; and
- 4) an error that will be `nil` if the character is syntactically valid.

The second argument, `quote`, specifies the type of literal being parsed and therefore which escaped quote character is permitted. If set to a single quote, it permits the sequence `\'` and disallows unescaped `'`. If set to a double quote, it permits `\"` and disallows unescaped `"`. If set to zero, it does not permit either escape and allows both quote characters to appear unescaped.

type [NumError](#)

```
type NumError struct {  
    Func string // the failing function (ParseBool, ParseInt, ParseU  
    Num  string // the input  
    Err  error   // the reason the conversion failed (ErrRange, ErrSy  
}
```

A NumError records a failed conversion.

func (*NumError) [Error](#)

```
func (e *NumError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package strings

```
import "strings"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package strings implements simple functions to manipulate strings.

Index

[func Contains\(s, substr string\) bool](#)
[func ContainsAny\(s, chars string\) bool](#)
[func ContainsRune\(s string, r rune\) bool](#)
[func Count\(s, sep string\) int](#)
[func EqualFold\(s, t string\) bool](#)
[func Fields\(s string\) \[\]string](#)
[func FieldsFunc\(s string, f func\(rune\) bool\) \[\]string](#)
[func HasPrefix\(s, prefix string\) bool](#)
[func HasSuffix\(s, suffix string\) bool](#)
[func Index\(s, sep string\) int](#)
[func IndexAny\(s, chars string\) int](#)
[func IndexFunc\(s string, f func\(rune\) bool\) int](#)
[func IndexRune\(s string, r rune\) int](#)
[func Join\(a \[\]string, sep string\) string](#)
[func LastIndex\(s, sep string\) int](#)
[func LastIndexAny\(s, chars string\) int](#)
[func LastIndexFunc\(s string, f func\(rune\) bool\) int](#)
[func Map\(mapping func\(rune\) rune, s string\) string](#)
[func Repeat\(s string, count int\) string](#)
[func Replace\(s, old, new string, n int\) string](#)
[func Split\(s, sep string\) \[\]string](#)
[func SplitAfter\(s, sep string\) \[\]string](#)
[func SplitAfterN\(s, sep string, n int\) \[\]string](#)
[func SplitN\(s, sep string, n int\) \[\]string](#)
[func Title\(s string\) string](#)
[func ToLower\(s string\) string](#)
[func ToLowerSpecial\(_ case unicode.SpecialCase, s string\) string](#)
[func ToTitle\(s string\) string](#)
[func ToTitleSpecial\(_ case unicode.SpecialCase, s string\) string](#)
[func ToUpper\(s string\) string](#)
[func ToUpperSpecial\(_ case unicode.SpecialCase, s string\) string](#)
[func Trim\(s string, cutset string\) string](#)
[func TrimFunc\(s string, f func\(rune\) bool\) string](#)
[func TrimLeft\(s string, cutset string\) string](#)
[func TrimLeftFunc\(s string, f func\(rune\) bool\) string](#)

[func TrimRight\(s string, cutset string\) string](#)
[func TrimRightFunc\(s string, f func\(rune\) bool\) string](#)
[func TrimSpace\(s string\) string](#)
[type Reader](#)
 [func NewReader\(s string\) *Reader](#)
 [func \(r *Reader\) Len\(\) int](#)
 [func \(r *Reader\) Read\(b \[\]byte\) \(n int, err error\)](#)
 [func \(r *Reader\) ReadAt\(b \[\]byte, off int64\) \(n int, err error\)](#)
 [func \(r *Reader\) ReadByte\(\) \(b byte, err error\)](#)
 [func \(r *Reader\) ReadRune\(\) \(ch rune, size int, err error\)](#)
 [func \(r *Reader\) Seek\(offset int64, whence int\) \(int64, error\)](#)
 [func \(r *Reader\) UnreadByte\(\) error](#)
 [func \(r *Reader\) UnreadRune\(\) error](#)
[type Replacer](#)
 [func NewReplacer\(oldnew ...string\) *Replacer](#)
 [func \(r *Replacer\) Replace\(s string\) string](#)
 [func \(r *Replacer\) WriteString\(w io.Writer, s string\) \(n int, err error\)](#)
[Bugs](#)

Examples

[Contains](#)
[ContainsAny](#)
[Count](#)
[EqualFold](#)
[Fields](#)
[Index](#)
[IndexRune](#)
[Join](#)
[LastIndex](#)
[Map](#)
[NewReplacer](#)
[Repeat](#)
[Replace](#)
[Split](#)
[SplitAfter](#)
[SplitAfterN](#)
[SplitN](#)

[Title](#)
[ToLower](#)
[ToTitle](#)
[ToUpper](#)
[Trim](#)
[TrimSpace](#)

Package files

[reader.go](#) [replace.go](#) [strings.go](#)

func Contains

func Contains(s, substr string) bool

Contains returns true if substr is within s.

? Example

? Example

Code:

```
fmt.Println(strings.Contains("seafood", "foo"))
fmt.Println(strings.Contains("seafood", "bar"))
fmt.Println(strings.Contains("seafood", ""))
fmt.Println(strings.Contains("", ""))
```

Output:

```
true
false
true
true
```

func ContainsAny

```
func ContainsAny(s, chars string) bool
```

ContainsAny returns true if any Unicode code points in chars are within s.

? Example

? Example

Code:

```
fmt.Println(strings.ContainsAny("team", "i"))  
fmt.Println(strings.ContainsAny("failure", "u & i"))  
fmt.Println(strings.ContainsAny("foo", ""))  
fmt.Println(strings.ContainsAny("", ""))
```

Output:

```
false  
true  
false  
false
```

func ContainsRune

```
func ContainsRune(s string, r rune) bool
```

ContainsRune returns true if the Unicode code point `r` is within `s`.

func [Count](#)

```
func Count(s, sep string) int
```

Count counts the number of non-overlapping instances of sep in s.

? Example

? Example

Code:

```
fmt.Println(strings.Count("cheese", "e"))  
fmt.Println(strings.Count("five", "")) // before & after each rune
```

Output:

```
3  
5
```

func [EqualFold](#)

```
func EqualFold(s, t string) bool
```

EqualFold reports whether s and t, interpreted as UTF-8 strings, are equal under Unicode case-folding.

? Example

? Example

Code:

```
fmt.Println(strings.EqualFold("Go", "go"))
```

Output:

```
true
```

func [Fields](#)

```
func Fields(s string) []string
```

Fields splits the string `s` around each instance of one or more consecutive white space characters, returning an array of substrings of `s` or an empty list if `s` contains only white space.

? Example

? Example

Code:

```
fmt.Printf("Fields are: %q", strings.Fields("  foo bar  baz  "))
```

Output:

```
Fields are: ["foo" "bar" "baz"]
```

func [FieldsFunc](#)

```
func FieldsFunc(s string, f func(rune) bool) []string
```

FieldsFunc splits the string `s` at each run of Unicode code points `c` satisfying `f(c)` and returns an array of slices of `s`. If all code points in `s` satisfy `f(c)` or the string is empty, an empty slice is returned.

func [HasPrefix](#)

```
func HasPrefix(s, prefix string) bool
```

HasPrefix tests whether the string `s` begins with `prefix`.

func [HasSuffix](#)

```
func HasSuffix(s, suffix string) bool
```

HasSuffix tests whether the string `s` ends with `suffix`.

func [Index](#)

func Index(s, sep string) int

Index returns the index of the first instance of sep in s, or -1 if sep is not present in s.

? Example

? Example

Code:

```
fmt.Println(strings.Index("chicken", "ken"))  
fmt.Println(strings.Index("chicken", "dmr"))
```

Output:

```
4  
-1
```

func [IndexAny](#)

```
func IndexAny(s, chars string) int
```

IndexAny returns the index of the first instance of any Unicode code point from chars in s, or -1 if no Unicode code point from chars is present in s.

func [IndexFunc](#)

```
func IndexFunc(s string, f func(rune) bool) int
```

IndexFunc returns the index into s of the first Unicode code point satisfying f(c), or -1 if none do.

func [IndexRune](#)

```
func IndexRune(s string, r rune) int
```

IndexRune returns the index of the first instance of the Unicode code point `r`, or `-1` if `rune` is not present in `s`.

? Example

? Example

Code:

```
fmt.Println(strings.IndexRune("chicken", 'k'))  
fmt.Println(strings.IndexRune("chicken", 'd'))
```

Output:

```
4  
-1
```

func [Join](#)

```
func Join(a []string, sep string) string
```

Join concatenates the elements of a to create a single string. The separator string sep is placed between elements in the resulting string.

? Example

? Example

Code:

```
s := []string{"foo", "bar", "baz"}  
fmt.Println(strings.Join(s, ", "))
```

Output:

```
foo, bar, baz
```

func [LastIndex](#)

```
func LastIndex(s, sep string) int
```

LastIndex returns the index of the last instance of sep in s, or -1 if sep is not present in s.

? Example

? Example

Code:

```
fmt.Println(strings.Index("go gopher", "go"))  
fmt.Println(strings.LastIndex("go gopher", "go"))  
fmt.Println(strings.LastIndex("go gopher", "rodent"))
```

Output:

```
0  
3  
-1
```

func [LastIndexAny](#)

```
func LastIndexAny(s, chars string) int
```

LastIndexAny returns the index of the last instance of any Unicode code point from chars in s, or -1 if no Unicode code point from chars is present in s.

func [LastIndexFunc](#)

```
func LastIndexFunc(s string, f func(rune) bool) int
```

LastIndexFunc returns the index into s of the last Unicode code point satisfying f(c), or -1 if none do.

func Map

```
func Map(mapping func(rune) rune, s string) string
```

Map returns a copy of the string `s` with all its characters modified according to the mapping function. If mapping returns a negative value, the character is dropped from the string with no replacement.

? Example

? Example

Code:

```
rot13 := func(r rune) rune {
    switch {
    case r >= 'A' && r <= 'Z':
        return 'A' + (r-'A'+13)%26
    case r >= 'a' && r <= 'z':
        return 'a' + (r-'a'+13)%26
    }
    return r
}
fmt.Println(strings.Map(rot13, "'Twas brillig and the slithy gopher.
```

Output:

```
'Gjnf oevyyvt naq gur fyvgul tbcure...
```

func Repeat

```
func Repeat(s string, count int) string
```

Repeat returns a new string consisting of count copies of the string s.

? Example

? Example

Code:

```
fmt.Println("ba" + strings.Repeat("na", 2))
```

Output:

banana

func [Replace](#)

```
func Replace(s, old, new string, n int) string
```

Replace returns a copy of the string `s` with the first `n` non-overlapping instances of `old` replaced by `new`. If `n < 0`, there is no limit on the number of replacements.

? Example

? Example

Code:

```
fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))  
fmt.Println(strings.Replace("oink oink oink", "oink", "moo", -1))
```

Output:

```
oinky oinky oink  
moo moo moo
```

func [Split](#)

```
func Split(s, sep string) []string
```

Split slices `s` into all substrings separated by `sep` and returns a slice of the substrings between those separators. If `sep` is empty, Split splits after each UTF-8 sequence. It is equivalent to SplitN with a count of -1.

? Example

? Example

Code:

```
fmt.Printf("%q\n", strings.Split("a,b,c", ","))
fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a "))
fmt.Printf("%q\n", strings.Split(" xyz ", ""))
fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
```

Output:

```
["a" "b" "c"]
["" "man " "plan " "canal panama"]
[" " "x" "y" "z" " "]
[""]
```

func [SplitAfter](#)

```
func SplitAfter(s, sep string) []string
```

`SplitAfter` slices `s` into all substrings after each instance of `sep` and returns a slice of those substrings. If `sep` is empty, `SplitAfter` splits after each UTF-8 sequence. It is equivalent to `SplitAfterN` with a count of `-1`.

? Example

? Example

Code:

```
fmt.Printf("%q\n", strings.SplitAfter("a,b,c", ","))
```

Output:

```
["a," "b," "c"]
```

func [SplitAfterN](#)

```
func SplitAfterN(s, sep string, n int) []string
```

SplitAfterN slices s into substrings after each instance of sep and returns a slice of those substrings. If sep is empty, SplitAfterN splits after each UTF-8 sequence. The count determines the number of substrings to return:

n > 0: at most n substrings; the last substring will be the unsplit

n == 0: the result is nil (zero substrings)

n < 0: all substrings

? Example

? Example

Code:

```
fmt.Printf("%q\n", strings.SplitAfterN("a,b,c", ",", 2))
```

Output:

```
["a," "b,c"]
```

func [SplitN](#)

```
func SplitN(s, sep string, n int) []string
```

SplitN slices s into substrings separated by sep and returns a slice of the substrings between those separators. If sep is empty, SplitN splits after each UTF-8 sequence. The count determines the number of substrings to return:

n > 0: at most n substrings; the last substring will be the unsplit
n == 0: the result is nil (zero substrings)
n < 0: all substrings

? Example

? Example

Code:

```
fmt.Printf("%q\n", strings.SplitN("a,b,c", ",", 2))  
z := strings.SplitN("a,b,c", ",", 0)  
fmt.Printf("%q (nil = %v)\n", z, z == nil)
```

Output:

```
["a" "b,c"]  
[] (nil = true)
```

func [Title](#)

```
func Title(s string) string
```

Title returns a copy of the string `s` with all Unicode letters that begin words mapped to their title case.

? Example

? Example

Code:

```
fmt.Println(strings.Title("her royal highness"))
```

Output:

```
Her Royal Highness
```

func [ToLower](#)

```
func ToLower(s string) string
```

ToLower returns a copy of the string `s` with all Unicode letters mapped to their lower case.

? Example

? Example

Code:

```
fmt.Println(strings.ToLower("Gopher"))
```

Output:

```
gopher
```

func [ToLowerSpecial](#)

```
func ToLowerSpecial(_case unicode.SpecialCase, s string) string
```

ToLowerSpecial returns a copy of the string `s` with all Unicode letters mapped to their lower case, giving priority to the special casing rules.

func [ToTitle](#)

```
func ToTitle(s string) string
```

ToTitle returns a copy of the string s with all Unicode letters mapped to their title case.

? Example

? Example

Code:

```
fmt.Println(strings.ToTitle("loud noises"))  
fmt.Println(strings.ToTitle("ᵀ"))
```

Output:

```
LOUD NOISES
```

func [ToTitleSpecial](#)

```
func ToTitleSpecial(_case unicode.SpecialCase, s string) string
```

ToTitleSpecial returns a copy of the string `s` with all Unicode letters mapped to their title case, giving priority to the special casing rules.

func [ToUpper](#)

```
func ToUpper(s string) string
```

ToUpper returns a copy of the string `s` with all Unicode letters mapped to their upper case.

? Example

? Example

Code:

```
fmt.Println(strings.ToUpper("Gopher"))
```

Output:

```
GOPHER
```

func [ToUpperSpecial](#)

```
func ToUpperSpecial(_case unicode.SpecialCase, s string) string
```

ToUpperSpecial returns a copy of the string `s` with all Unicode letters mapped to their upper case, giving priority to the special casing rules.

func [Trim](#)

```
func Trim(s string, cutset string) string
```

Trim returns a slice of the string `s` with all leading and trailing Unicode code points contained in `cutset` removed.

? Example

? Example

Code:

```
fmt.Printf("[%q]", strings.Trim(" !!! Achtung !!! ", "! "))
```

Output:

```
["Achtung"]
```

func [TrimFunc](#)

```
func TrimFunc(s string, f func(rune) bool) string
```

TrimFunc returns a slice of the string s with all leading and trailing Unicode code points c satisfying f(c) removed.

func [TrimLeft](#)

```
func TrimLeft(s string, cutset string) string
```

TrimLeft returns a slice of the string s with all leading Unicode code points contained in cutset removed.

func [TrimLeftFunc](#)

```
func TrimLeftFunc(s string, f func(rune) bool) string
```

TrimLeftFunc returns a slice of the string `s` with all leading Unicode code points `c` satisfying `f(c)` removed.

func [TrimRight](#)

```
func TrimRight(s string, cutset string) string
```

TrimRight returns a slice of the string s, with all trailing Unicode code points contained in cutset removed.

func [TrimRightFunc](#)

```
func TrimRightFunc(s string, f func(rune) bool) string
```

TrimRightFunc returns a slice of the string s with all trailing Unicode code points c satisfying f(c) removed.

func [TrimSpace](#)

```
func TrimSpace(s string) string
```

TrimSpace returns a slice of the string s, with all leading and trailing white space removed, as defined by Unicode.

? Example

? Example

Code:

```
fmt.Println(strings.TrimSpace(" \t\n a lone gopher \n\t\r\n"))
```

Output:

```
a lone gopher
```

type [Reader](#)

```
type Reader struct {  
    // contains filtered or unexported fields  
}
```

A Reader implements the `io.Reader`, `io.ReaderAt`, `io.Seeker`, `io.ByteScanner`, and `io.RuneScanner` interfaces by reading from a string.

func [NewReader](#)

```
func NewReader(s string) *Reader
```

`NewReader` returns a new Reader reading from `s`. It is similar to `bytes.NewBufferString` but more efficient and read-only.

func (***Reader**) [Len](#)

```
func (r *Reader) Len() int
```

`Len` returns the number of bytes of the unread portion of the string.

func (***Reader**) [Read](#)

```
func (r *Reader) Read(b []byte) (n int, err error)
```

func (***Reader**) [ReadAt](#)

```
func (r *Reader) ReadAt(b []byte, off int64) (n int, err error)
```

func (***Reader**) [ReadByte](#)

```
func (r *Reader) ReadByte() (b byte, err error)
```

func (***Reader**) [ReadRune](#)

```
func (r *Reader) ReadRune() (ch rune, size int, err error)
```

func (***Reader**) [Seek](#)

```
func (r *Reader) Seek(offset int64, whence int) (int64, error)
```

Seek implements the io.Seeker interface.

func (*Reader) [UnreadByte](#)

```
func (r *Reader) UnreadByte() error
```

func (*Reader) [UnreadRune](#)

```
func (r *Reader) UnreadRune() error
```

type [Replacer](#)

```
type Replacer struct {  
    // contains filtered or unexported fields  
}
```

A Replacer replaces a list of strings with replacements.

func [NewReplacer](#)

```
func NewReplacer(oldnew ...string) *Replacer
```

NewReplacer returns a new Replacer from a list of old, new string pairs. Replacements are performed in order, without overlapping matches.

? Example

? Example

Code:

```
r := strings.NewReplacer("<", "&lt;", ">", "&gt;")  
fmt.Println(r.Replace("This is <b>HTML</b>!"))
```

Output:

```
This is &lt;b&gt;HTML&lt;/b&gt;!
```

func (*Replacer) [Replace](#)

```
func (r *Replacer) Replace(s string) string
```

Replace returns a copy of s with all replacements performed.

func (*Replacer) [WriteString](#)

```
func (r *Replacer) WriteString(w io.Writer, s string) (n int, err error)
```

WriteString writes s to w with all replacements performed.

Bugs

The rule Title uses for word boundaries does not handle Unicode punctuation properly.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package sync

```
import "sync"
```

[Overview](#)

[Index](#)

[Examples](#)

[Subdirectories](#)

Overview ?

Overview ?

Package sync provides basic synchronization primitives such as mutual exclusion locks. Other than the Once and WaitGroup types, most are intended for use by low-level library routines. Higher-level synchronization is better done via channels and communication.

Values containing the types defined in this package should not be copied.

Index

[type Cond](#)

[func NewCond\(l Locker\) *Cond](#)

[func \(c *Cond\) Broadcast\(\)](#)

[func \(c *Cond\) Signal\(\)](#)

[func \(c *Cond\) Wait\(\)](#)

[type Locker](#)

[type Mutex](#)

[func \(m *Mutex\) Lock\(\)](#)

[func \(m *Mutex\) Unlock\(\)](#)

[type Once](#)

[func \(o *Once\) Do\(f func\(\)\)](#)

[type RWMutex](#)

[func \(rw *RWMutex\) Lock\(\)](#)

[func \(rw *RWMutex\) RLock\(\)](#)

[func \(rw *RWMutex\) RLocker\(\) Locker](#)

[func \(rw *RWMutex\) RUnlock\(\)](#)

[func \(rw *RWMutex\) Unlock\(\)](#)

[type WaitGroup](#)

[func \(wg *WaitGroup\) Add\(delta int\)](#)

[func \(wg *WaitGroup\) Done\(\)](#)

[func \(wg *WaitGroup\) Wait\(\)](#)

Examples

[Once](#)

[WaitGroup](#)

Package files

[cond.go](#) [mutex.go](#) [once.go](#) [runtime.go](#) [rwmutex.go](#) [waitgroup.go](#)

type [Cond](#)

```
type Cond struct {  
    L Locker // held while observing or changing the condition  
    // contains filtered or unexported fields  
}
```

Cond implements a condition variable, a rendezvous point for goroutines waiting for or announcing the occurrence of an event.

Each Cond has an associated Locker L (often a *Mutex or *RWMutex), which must be held when changing the condition and when calling the Wait method.

func [NewCond](#)

```
func NewCond(l Locker) *Cond
```

NewCond returns a new Cond with Locker l.

func (***Cond**) [Broadcast](#)

```
func (c *Cond) Broadcast()
```

Broadcast wakes all goroutines waiting on c.

It is allowed but not required for the caller to hold c.L during the call.

func (***Cond**) [Signal](#)

```
func (c *Cond) Signal()
```

Signal wakes one goroutine waiting on c, if there is any.

It is allowed but not required for the caller to hold c.L during the call.

func (***Cond**) [Wait](#)

```
func (c *Cond) Wait()
```

Wait atomically unlocks c.L and suspends execution of the calling goroutine. After later resuming execution, Wait locks c.L before returning. Unlike in other systems, Wait cannot return unless awoken by Broadcast or Signal.

Because c.L is not locked when Wait first resumes, the caller typically cannot assume that the condition is true when Wait returns. Instead, the caller should Wait in a loop:

```
c.L.Lock()
for !condition() {
    c.Wait()
}
... make use of condition ...
c.L.Unlock()
```

type Locker

```
type Locker interface {  
    Lock()  
    Unlock()  
}
```

A Locker represents an object that can be locked and unlocked.

type [Mutex](#)

```
type Mutex struct {  
    // contains filtered or unexported fields  
}
```

A Mutex is a mutual exclusion lock. Mutexes can be created as part of other structures; the zero value for a Mutex is an unlocked mutex.

func (*Mutex) [Lock](#)

```
func (m *Mutex) Lock()
```

Lock locks m. If the lock is already in use, the calling goroutine blocks until the mutex is available.

func (*Mutex) [Unlock](#)

```
func (m *Mutex) Unlock()
```

Unlock unlocks m. It is a run-time error if m is not locked on entry to Unlock.

A locked Mutex is not associated with a particular goroutine. It is allowed for one goroutine to lock a Mutex and then arrange for another goroutine to unlock it.

type [Once](#)

```
type Once struct {  
    // contains filtered or unexported fields  
}
```

Once is an object that will perform exactly one action.

? Example

? Example

Code:

```
var once sync.Once  
onceBody := func() {  
    fmt.Printf("Only once\n")  
}  
done := make(chan bool)  
for i := 0; i < 10; i++ {  
    go func() {  
        once.Do(onceBody)  
        done <- true  
    }()  
}  
for i := 0; i < 10; i++ {  
    <-done  
}
```

Output:

Only once

func (*Once) [Do](#)

```
func (o *Once) Do(f func())
```

Do calls the function f if and only if the method is being called for the first time with this receiver. In other words, given

```
var once Once
```

if `once.Do(f)` is called multiple times, only the first call will invoke f, even if f

has a different value in each invocation. A new instance of `Once` is required for each function to execute.

`Do` is intended for initialization that must be run exactly once. Since `f` is niladic, it may be necessary to use a function literal to capture the arguments to a function to be invoked by `Do`:

```
config.once.Do(func() { config.init(filename) })
```

Because no call to `Do` returns until the one call to `f` returns, if `f` causes `Do` to be called, it will deadlock.

type [RWMutex](#)

```
type RWMutex struct {  
    // contains filtered or unexported fields  
}
```

An RWMutex is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer. RWMutexes can be created as part of other structures; the zero value for a RWMutex is an unlocked mutex.

func (*RWMutex) [Lock](#)

```
func (rw *RWMutex) Lock()
```

Lock locks rw for writing. If the lock is already locked for reading or writing, Lock blocks until the lock is available. To ensure that the lock eventually becomes available, a blocked Lock call excludes new readers from acquiring the lock.

func (*RWMutex) [RLock](#)

```
func (rw *RWMutex) RLock()
```

RLock locks rw for reading.

func (*RWMutex) [RLocker](#)

```
func (rw *RWMutex) RLocker() Locker
```

RLocker returns a Locker interface that implements the Lock and Unlock methods by calling rw.RLock and rw.RUnlock.

func (*RWMutex) [RUnlock](#)

```
func (rw *RWMutex) RUnlock()
```

RUnlock undoes a single RLock call; it does not affect other simultaneous readers. It is a run-time error if rw is not locked for reading on entry to RUnlock.

func (*RWMutex) [Unlock](#)

```
func (rw *RWMutex) Unlock()
```

Unlock unlocks rw for writing. It is a run-time error if rw is not locked for writing on entry to Unlock.

As with Mutexes, a locked RWMutex is not associated with a particular goroutine. One goroutine may RLock (Lock) an RWMutex and then arrange for another goroutine to RUnlock (Unlock) it.

type [WaitGroup](#)

```
type WaitGroup struct {  
    // contains filtered or unexported fields  
}
```

A `WaitGroup` waits for a collection of goroutines to finish. The main goroutine calls `Add` to set the number of goroutines to wait for. Then each of the goroutines runs and calls `Done` when finished. At the same time, `Wait` can be used to block until all goroutines have finished.

? Example

? Example

This example fetches several URLs concurrently, using a `WaitGroup` to block until all the fetches are complete.

Code:

```
var wg sync.WaitGroup  
var urls = []string{  
    "http://www.golang.org/",  
    "http://www.google.com/",  
    "http://www.somestupidname.com/",  
}  
for _, url := range urls {  
    // Increment the WaitGroup counter.  
    wg.Add(1)  
    // Launch a goroutine to fetch the URL.  
    go func(url string) {  
        // Fetch the URL.  
        http.Get(url)  
        // Decrement the counter.  
        wg.Done()  
    }(url)  
}  
// Wait for all HTTP fetches to complete.  
wg.Wait()
```

func (*[WaitGroup](#)) [Add](#)

```
func (wg *WaitGroup) Add(delta int)
```

Add adds delta, which may be negative, to the WaitGroup counter. If the counter becomes zero, all goroutines blocked on Wait() are released.

func (*WaitGroup) [Done](#)

```
func (wg *WaitGroup) Done()
```

Done decrements the WaitGroup counter.

func (*WaitGroup) [Wait](#)

```
func (wg *WaitGroup) Wait()
```

Wait blocks until the WaitGroup counter is zero.

Subdirectories

Name **Synopsis**

[atomic](#) Package atomic provides low-level atomic memory primitives useful for implementing synchronization algorithms.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package atomic

```
import "sync/atomic"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `atomic` provides low-level atomic memory primitives useful for implementing synchronization algorithms.

These functions require great care to be used correctly. Except for special, low-level applications, synchronization is better done with channels or the facilities of the `sync` package. Share memory by communicating; don't communicate by sharing memory.

The compare-and-swap operation, implemented by the `CompareAndSwapT` functions, is the atomic equivalent of:

```
if *val == old {
    *val = new
    return true
}
return false
```

Index

[func AddInt32\(val *int32, delta int32\) \(new int32\)](#)
[func AddInt64\(val *int64, delta int64\) \(new int64\)](#)
[func AddUint32\(val *uint32, delta uint32\) \(new uint32\)](#)
[func AddUint64\(val *uint64, delta uint64\) \(new uint64\)](#)
[func AddUintptr\(val *uintptr, delta uintptr\) \(new uintptr\)](#)
[func CompareAndSwapInt32\(val *int32, old, new int32\) \(swapped bool\)](#)
[func CompareAndSwapInt64\(val *int64, old, new int64\) \(swapped bool\)](#)
[func CompareAndSwapPointer\(val *unsafe.Pointer, old, new unsafe.Pointer\) \(swapped bool\)](#)
[func CompareAndSwapUint32\(val *uint32, old, new uint32\) \(swapped bool\)](#)
[func CompareAndSwapUint64\(val *uint64, old, new uint64\) \(swapped bool\)](#)
[func CompareAndSwapUintptr\(val *uintptr, old, new uintptr\) \(swapped bool\)](#)
[func LoadInt32\(addr *int32\) \(val int32\)](#)
[func LoadInt64\(addr *int64\) \(val int64\)](#)
[func LoadPointer\(addr *unsafe.Pointer\) \(val unsafe.Pointer\)](#)
[func LoadUint32\(addr *uint32\) \(val uint32\)](#)
[func LoadUint64\(addr *uint64\) \(val uint64\)](#)
[func LoadUintptr\(addr *uintptr\) \(val uintptr\)](#)
[func StoreInt32\(addr *int32, val int32\)](#)
[func StoreInt64\(addr *int64, val int64\)](#)
[func StorePointer\(addr *unsafe.Pointer, val unsafe.Pointer\)](#)
[func StoreUint32\(addr *uint32, val uint32\)](#)
[func StoreUint64\(addr *uint64, val uint64\)](#)
[func StoreUintptr\(addr *uintptr, val uintptr\)](#)
[Bugs](#)

Package files

doc.go

func [AddInt32](#)

```
func AddInt32(val *int32, delta int32) (new int32)
```

AddInt32 atomically adds delta to *val and returns the new value.

func [AddInt64](#)

```
func AddInt64(val *int64, delta int64) (new int64)
```

AddInt64 atomically adds delta to *val and returns the new value.

func [AddUint32](#)

```
func AddUint32(val *uint32, delta uint32) (new uint32)
```

AddUint32 atomically adds delta to *val and returns the new value.

func [AddUint64](#)

```
func AddUint64(val *uint64, delta uint64) (new uint64)
```

AddUint64 atomically adds delta to *val and returns the new value.

func [AddUintptr](#)

```
func AddUintptr(val *uintptr, delta uintptr) (new uintptr)
```

AddUintptr atomically adds delta to *val and returns the new value.

func [CompareAndSwapInt32](#)

```
func CompareAndSwapInt32(val *int32, old, new int32) (swapped bool)
```

CompareAndSwapInt32 executes the compare-and-swap operation for an int32 value.

func [CompareAndSwapInt64](#)

```
func CompareAndSwapInt64(val *int64, old, new int64) (swapped bool)
```

CompareAndSwapInt64 executes the compare-and-swap operation for an int64 value.

func CompareAndSwapPointer

```
func CompareAndSwapPointer(val *unsafe.Pointer, old, new unsafe.Poin
```

CompareAndSwapPointer executes the compare-and-swap operation for a `unsafe.Pointer` value.

func [CompareAndSwapUint32](#)

`func CompareAndSwapUint32(val *uint32, old, new uint32) (swapped bool)`

`CompareAndSwapUint32` executes the compare-and-swap operation for a `uint32` value.

func [CompareAndSwapUint64](#)

```
func CompareAndSwapUint64(val *uint64, old, new uint64) (swapped bool)
```

CompareAndSwapUint64 executes the compare-and-swap operation for a uint64 value.

func CompareAndSwapUintptr

```
func CompareAndSwapUintptr(val *uintptr, old, new uintptr) (swapped
```

CompareAndSwapUintptr executes the compare-and-swap operation for a uintptr value.

func LoadInt32

```
func LoadInt32(addr *int32) (val int32)
```

LoadInt32 atomically loads *addr.

func [LoadInt64](#)

```
func LoadInt64(addr *int64) (val int64)
```

LoadInt64 atomically loads *addr.

func LoadPointer

```
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
```

LoadPointer atomically loads *addr.

func [LoadUint32](#)

```
func LoadUint32(addr *uint32) (val uint32)
```

LoadUint32 atomically loads *addr.

func [LoadUint64](#)

```
func LoadUint64(addr *uint64) (val uint64)
```

LoadUint64 atomically loads *addr.

func LoadUintptr

```
func LoadUintptr(addr *uintptr) (val uintptr)
```

LoadUintptr atomically loads *addr.

func [StoreInt32](#)

```
func StoreInt32(addr *int32, val int32)
```

StoreInt32 atomically stores val into *addr.

func [StoreInt64](#)

```
func StoreInt64(addr *int64, val int64)
```

StoreInt64 atomically stores val into *addr.

func StorePointer

```
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
```

StorePointer atomically stores val into *addr.

func [StoreUint32](#)

```
func StoreUint32(addr *uint32, val uint32)
```

StoreUint32 atomically stores val into *addr.

func [StoreUint64](#)

```
func StoreUint64(addr *uint64, val uint64)
```

StoreUint64 atomically stores val into *addr.

func [StoreUintptr](#)

```
func StoreUintptr(addr *uintptr, val uintptr)
```

StoreUintptr atomically stores val into *addr.

Bugs

On ARM, the 64-bit functions use instructions unavailable before ARM 11.

On x86-32, the 64-bit functions use instructions unavailable before the Pentium MMX.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package syscall

```
import "syscall"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `syscall` contains an interface to the low-level operating system primitives. The details vary depending on the underlying system. Its primary use is inside other packages that provide a more portable interface to the system, such as `"os"`, `"time"` and `"net"`. Use those packages rather than this one if you can. For details of the functions and data types in this package consult the manuals for the appropriate operating system. These calls return `err == nil` to indicate success; otherwise `err` is an operating system error describing the failure. On most systems, that error has type `syscall.Errno`.

Index

Constants

Variables

[func Accept\(fd int\) \(nfd int, sa Sockaddr, err error\)](#)

[func Access\(path string, mode uint32\) \(err error\)](#)

[func Acct\(path string\) \(err error\)](#)

[func Adjtimex\(buf *Timex\) \(state int, err error\)](#)

[func AttachLsf\(fd int, i \[\]SockFilter\) error](#)

[func Bind\(fd int, sa Sockaddr\) \(err error\)](#)

[func BindToDevice\(fd int, device string\) \(err error\)](#)

[func Chdir\(path string\) \(err error\)](#)

[func Chmod\(path string, mode uint32\) \(err error\)](#)

[func Chown\(path string, uid int, gid int\) \(err error\)](#)

[func Chroot\(path string\) \(err error\)](#)

[func Clearenv\(\)](#)

[func Close\(fd int\) \(err error\)](#)

[func CloseOnExec\(fd int\)](#)

[func CmsgLen\(datalen int\) int](#)

[func CmsgSpace\(datalen int\) int](#)

[func Connect\(fd int, sa Sockaddr\) \(err error\)](#)

[func Creat\(path string, mode uint32\) \(fd int, err error\)](#)

[func DetachLsf\(fd int\) error](#)

[func Dup\(oldfd int\) \(fd int, err error\)](#)

[func Dup2\(oldfd int, newfd int\) \(err error\)](#)

[func Environ\(\) \[\]string](#)

[func EpollCreate\(size int\) \(fd int, err error\)](#)

[func EpollCreate1\(flag int\) \(fd int, err error\)](#)

[func EpollCtl\(epfd int, op int, fd int, event *EpollEvent\) \(err error\)](#)

[func EpollWait\(epfd int, events \[\]EpollEvent, msec int\) \(n int, err error\)](#)

[func Exec\(argv0 string, argv \[\]string, envv \[\]string\) \(err error\)](#)

[func Exit\(code int\)](#)

[func Faccessat\(dirfd int, path string, mode uint32, flags int\) \(err error\)](#)

[func Fallocate\(fd int, mode uint32, off int64, len int64\) \(err error\)](#)

[func Fchdir\(fd int\) \(err error\)](#)

[func Fchmod\(fd int, mode uint32\) \(err error\)](#)

[func Fchmodat\(dirfd int, path string, mode uint32, flags int\) \(err error\)](#)

[func Fchown\(fd int, uid int, gid int\) \(err error\)](#)
[func Fchownat\(dirfd int, path string, uid int, gid int, flags int\) \(err error\)](#)
[func Fdatasync\(fd int\) \(err error\)](#)
[func Flock\(fd int, how int\) \(err error\)](#)
[func ForkExec\(argv0 string, argv \[\]string, attr *ProcAttr\) \(pid int, err error\)](#)
[func Fstat\(fd int, stat *Stat_t\) \(err error\)](#)
[func Fstatfs\(fd int, buf *Statfs_t\) \(err error\)](#)
[func Fsync\(fd int\) \(err error\)](#)
[func Ftruncate\(fd int, length int64\) \(err error\)](#)
[func Futimes\(fd int, tv \[\]Timeval\) \(err error\)](#)
[func Futimesat\(dirfd int, path string, tv \[\]Timeval\) \(err error\)](#)
[func Getcwd\(buf \[\]byte\) \(n int, err error\)](#)
[func Getdents\(fd int, buf \[\]byte\) \(n int, err error\)](#)
[func Getegid\(\) \(egid int\)](#)
[func Getenv\(key string\) \(value string, found bool\)](#)
[func Geteuid\(\) \(euid int\)](#)
[func Getgid\(\) \(gid int\)](#)
[func Getgroups\(\) \(gids \[\]int, err error\)](#)
[func Getpagesize\(\) int](#)
[func Getpgid\(pid int\) \(pgid int, err error\)](#)
[func Getpgrp\(\) \(pid int\)](#)
[func Getpid\(\) \(pid int\)](#)
[func Getppid\(\) \(ppid int\)](#)
[func Getrlimit\(resource int, rlim *Rlimit\) \(err error\)](#)
[func Getrusage\(who int, rusage *Rusage\) \(err error\)](#)
[func GetsockoptInet4Addr\(fd, level, opt int\) \(value \[4\]byte, err error\)](#)
[func GetsockoptInt\(fd, level, opt int\) \(value int, err error\)](#)
[func Gettid\(\) \(tid int\)](#)
[func Gettimeofday\(tv *Timeval\) \(err error\)](#)
[func Getuid\(\) \(uid int\)](#)
[func Getwd\(\) \(wd string, err error\)](#)
[func InotifyAddWatch\(fd int, pathname string, mask uint32\) \(watchdesc int, err error\)](#)
[func InotifyInit\(\) \(fd int, err error\)](#)
[func InotifyInit1\(flags int\) \(fd int, err error\)](#)
[func InotifyRmWatch\(fd int, watchdesc uint32\) \(success int, err error\)](#)
[func Ioperm\(from int, num int, on int\) \(err error\)](#)
[func Iopl\(level int\) \(err error\)](#)
[func Kill\(pid int, sig Signal\) \(err error\)](#)

[func Klogctl\(typ int, buf \[\]byte\) \(n int, err error\)](#)
[func Lchown\(path string, uid int, gid int\) \(err error\)](#)
[func Link\(oldpath string, newpath string\) \(err error\)](#)
[func Listen\(s int, n int\) \(err error\)](#)
[func LsfSocket\(ifindex, proto int\) \(int, error\)](#)
[func Lstat\(path string, stat *Stat_t\) \(err error\)](#)
[func Madvise\(b \[\]byte, advice int\) \(err error\)](#)
[func Mkdir\(path string, mode uint32\) \(err error\)](#)
[func Mkdirat\(dirfd int, path string, mode uint32\) \(err error\)](#)
[func Mkfifo\(path string, mode uint32\) \(err error\)](#)
[func Mknod\(path string, mode uint32, dev int\) \(err error\)](#)
[func Mknodat\(dirfd int, path string, mode uint32, dev int\) \(err error\)](#)
[func Mlock\(b \[\]byte\) \(err error\)](#)
[func Mlockall\(flags int\) \(err error\)](#)
[func Mmap\(fd int, offset int64, length int, prot int, flags int\) \(data \[\]byte, err error\)](#)
[func Mount\(source string, target string, fstype string, flags uintptr, data string\) \(err error\)](#)
[func Mprotect\(b \[\]byte, prot int\) \(err error\)](#)
[func Munlock\(b \[\]byte\) \(err error\)](#)
[func Munlockall\(\) \(err error\)](#)
[func Munmap\(b \[\]byte\) \(err error\)](#)
[func Nanosleep\(time *Timespec, leftover *Timespec\) \(err error\)](#)
[func NetlinkRIB\(proto, family int\) \(\[\]byte, error\)](#)
[func Open\(path string, mode int, perm uint32\) \(fd int, err error\)](#)
[func Openat\(dirfd int, path string, flags int, mode uint32\) \(fd int, err error\)](#)
[func ParseDirent\(buf \[\]byte, max int, names \[\]string\) \(consumed int, count int, newnames \[\]string\)](#)
[func ParseNetlinkMessage\(buf \[\]byte\) \(\[\]NetlinkMessage, error\)](#)
[func ParseNetlinkRouteAttr\(msg *NetlinkMessage\) \(\[\]NetlinkRouteAttr, error\)](#)
[func ParseSocketControlMessage\(buf \[\]byte\) \(\[\]SocketControlMessage, error\)](#)
[func ParseUnixRights\(msg *SocketControlMessage\) \(\[\]int, error\)](#)
[func Pause\(\) \(err error\)](#)
[func Pipe\(p \[\]int\) \(err error\)](#)
[func PivotRoot\(newroot string, putold string\) \(err error\)](#)
[func Pread\(fd int, p \[\]byte, offset int64\) \(n int, err error\)](#)
[func PtraceAttach\(pid int\) \(err error\)](#)

[func PtraceCont\(pid int, signal int\) \(err error\)](#)
[func PtraceDetach\(pid int\) \(err error\)](#)
[func PtraceGetEventMsg\(pid int\) \(msg uint, err error\)](#)
[func PtraceGetRegs\(pid int, regsout *PtraceRegs\) \(err error\)](#)
[func PtracePeekData\(pid int, addr uintptr, out \[\]byte\) \(count int, err error\)](#)
[func PtracePeekText\(pid int, addr uintptr, out \[\]byte\) \(count int, err error\)](#)
[func PtracePokeData\(pid int, addr uintptr, data \[\]byte\) \(count int, err error\)](#)
[func PtracePokeText\(pid int, addr uintptr, data \[\]byte\) \(count int, err error\)](#)
[func PtraceSetOptions\(pid int, options int\) \(err error\)](#)
[func PtraceSetRegs\(pid int, regs *PtraceRegs\) \(err error\)](#)
[func PtraceSingleStep\(pid int\) \(err error\)](#)
[func Pwrite\(fd int, p \[\]byte, offset int64\) \(n int, err error\)](#)
[func RawSyscall\(trap, a1, a2, a3 uintptr\) \(r1, r2 uintptr, err Errno\)](#)
[func RawSyscall6\(trap, a1, a2, a3, a4, a5, a6 uintptr\) \(r1, r2 uintptr, err Errno\)](#)
[func Read\(fd int, p \[\]byte\) \(n int, err error\)](#)
[func ReadDirent\(fd int, buf \[\]byte\) \(n int, err error\)](#)
[func Readlink\(path string, buf \[\]byte\) \(n int, err error\)](#)
[func Reboot\(cmd int\) \(err error\)](#)
[func Recvfrom\(fd int, p \[\]byte, flags int\) \(n int, from Sockaddr, err error\)](#)
[func Recvmsg\(fd int, p, oob \[\]byte, flags int\) \(n, oobn int, recvflags int, from Sockaddr, err error\)](#)
[func Rename\(oldpath string, newpath string\) \(err error\)](#)
[func Renameat.olddirfd int, oldpath string, newdirfd int, newpath string\) \(err error\)](#)
[func Rmdir\(path string\) \(err error\)](#)
[func Seek\(fd int, offset int64, whence int\) \(off int64, err error\)](#)
[func Select\(nfd int, r *FdSet, w *FdSet, e *FdSet, timeout *Timeval\) \(n int, err error\)](#)
[func Sendfile\(outfd int, infd int, offset *int64, count int\) \(written int, err error\)](#)
[func Sendmsg\(fd int, p, oob \[\]byte, to Sockaddr, flags int\) \(err error\)](#)
[func Sendto\(fd int, p \[\]byte, flags int, to Sockaddr\) \(err error\)](#)
[func SetLsfPromisc\(name string, m bool\) error](#)
[func SetNonblock\(fd int, nonblocking bool\) \(err error\)](#)
[func Setdomainname\(p \[\]byte\) \(err error\)](#)
[func Setenv\(key, value string\) error](#)
[func Setfsuid\(gid int\) \(err error\)](#)
[func Setfsuid\(uid int\) \(err error\)](#)

[func Setgid\(gid int\) \(err error\)](#)
[func Setgroups\(gids \[\]int\) \(err error\)](#)
[func Sethostname\(p \[\]byte\) \(err error\)](#)
[func Setpgid\(pid int, pgid int\) \(err error\)](#)
[func Setregid\(rgid int, egid int\) \(err error\)](#)
[func Setresgid\(rgid int, egid int, sgid int\) \(err error\)](#)
[func Setresuid\(ruid int, euid int, suid int\) \(err error\)](#)
[func Setreuid\(ruid int, euid int\) \(err error\)](#)
[func Setrlimit\(resource int, rlim *Rlimit\) \(err error\)](#)
[func Setsid\(\) \(pid int, err error\)](#)
[func SetsockoptIPMreq\(fd, level, opt int, mreq *IPMreq\) \(err error\)](#)
[func SetsockoptIPMreqn\(fd, level, opt int, mreq *IPMreqn\) \(err error\)](#)
[func SetsockoptIPv6Mreq\(fd, level, opt int, mreq *IPv6Mreq\) \(err error\)](#)
[func SetsockoptInet4Addr\(fd, level, opt int, value \[4\]byte\) \(err error\)](#)
[func SetsockoptInt\(fd, level, opt int, value int\) \(err error\)](#)
[func SetsockoptLinger\(fd, level, opt int, l *Linger\) \(err error\)](#)
[func SetsockoptString\(fd, level, opt int, s string\) \(err error\)](#)
[func SetsockoptTimeval\(fd, level, opt int, tv *Timeval\) \(err error\)](#)
[func Settimeofday\(tv *Timeval\) \(err error\)](#)
[func Setuid\(uid int\) \(err error\)](#)
[func Shutdown\(fd int, how int\) \(err error\)](#)
[func Socket\(domain, typ, proto int\) \(fd int, err error\)](#)
[func Socketpair\(domain, typ, proto int\) \(fd \[2\]int, err error\)](#)
[func Splice\(rfd int, roff *int64, wfd int, woff *int64, len int, flags int\) \(n int64, err error\)](#)
[func StartProcess\(argv0 string, argv \[\]string, attr *ProcAttr\) \(pid int, handle uintptr, err error\)](#)
[func Stat\(path string, stat *Stat_t\) \(err error\)](#)
[func Statfs\(path string, buf *Statfs_t\) \(err error\)](#)
[func StringBytePtr\(s string\) *byte](#)
[func StringByteSlice\(s string\) \[\]byte](#)
[func StringSlicePtr\(ss \[\]string\) \[\]*byte](#)
[func Symlink\(oldpath string, newpath string\) \(err error\)](#)
[func Sync\(\)](#)
[func SyncFileRange\(fd int, off int64, n int64, flags int\) \(err error\)](#)
[func Syscall\(trap, a1, a2, a3 uintptr\) \(r1, r2 uintptr, err Errno\)](#)
[func Syscall6\(trap, a1, a2, a3, a4, a5, a6 uintptr\) \(r1, r2 uintptr, err Errno\)](#)
[func Sysinfo\(info *Sysinfo_t\) \(err error\)](#)
[func Tee\(rfd int, wfd int, len int, flags int\) \(n int64, err error\)](#)

[func Tgkill\(tgid int, tid int, sig Signal\) \(err error\)](#)
[func Times\(tms *Tms\) \(ticks uintptr, err error\)](#)
[func TimespecToNsec\(ts Timespec\) int64](#)
[func TimevalToNsec\(tv Timeval\) int64](#)
[func Truncate\(path string, length int64\) \(err error\)](#)
[func Umask\(mask int\) \(oldmask int\)](#)
[func Uname\(buf *Utsname\) \(err error\)](#)
[func UnixCredentials\(ucred *Ucred\) \[\]byte](#)
[func UnixRights\(fds ...int\) \[\]byte](#)
[func Unlink\(path string\) \(err error\)](#)
[func Unlinkat\(dirfd int, path string\) \(err error\)](#)
[func Unmount\(target string, flags int\) \(err error\)](#)
[func Unshare\(flags int\) \(err error\)](#)
[func Ustat\(dev int, ubuf *Ustat_t\) \(err error\)](#)
[func Utime\(path string, buf *Utimbuf\) \(err error\)](#)
[func Utimes\(path string, tv \[\]Timeval\) \(err error\)](#)
[func Wait4\(pid int, wstatus *WaitStatus, options int, rusage *Rusage\) \(wpid int, err error\)](#)
[func Write\(fd int, p \[\]byte\) \(n int, err error\)](#)
[type Cmsghdr](#)
 [func \(msg *Cmsghdr\) SetLen\(length int\)](#)
[type Credential](#)
[type Dirent](#)
[type EpollEvent](#)
[type Errno](#)
 [func \(e Errno\) Error\(\) string](#)
 [func \(e Errno\) Temporary\(\) bool](#)
 [func \(e Errno\) Timeout\(\) bool](#)
[type FdSet](#)
[type Fsid](#)
[type IPMreq](#)
 [func GetsockoptIPMreq\(fd, level, opt int\) \(*IPMreq, error\)](#)
[type IPMreqn](#)
 [func GetsockoptIPMreqn\(fd, level, opt int\) \(*IPMreqn, error\)](#)
[type IPv6Mreq](#)
 [func GetsockoptIPv6Mreq\(fd, level, opt int\) \(*IPv6Mreq, error\)](#)
[type IfAddrmsg](#)
[type IfInfomsg](#)
[type Inet4Pktinfo](#)

[type Inet6Pktinfo](#)
[type InotifyEvent](#)
[type Iovec](#)
 [func \(iov *Iovec\) SetLen\(length int\)](#)
[type Linger](#)
[type Msghdr](#)
 [func \(msghdr *Msghdr\) SetControllen\(length int\)](#)
[type NetlinkMessage](#)
[type NetlinkRouteAttr](#)
[type NetlinkRouteRequest](#)
[type NIAttr](#)
[type NIMsgerr](#)
[type NIMsghdr](#)
[type ProcAttr](#)
[type PtraceRegs](#)
 [func \(r *PtraceRegs\) PC\(\) uint64](#)
 [func \(r *PtraceRegs\) SetPC\(pc uint64\)](#)
[type RawSockaddr](#)
[type RawSockaddrAny](#)
[type RawSockaddrInet4](#)
[type RawSockaddrInet6](#)
[type RawSockaddrLinklayer](#)
[type RawSockaddrNetlink](#)
[type RawSockaddrUnix](#)
[type Rlimit](#)
[type RtAttr](#)
[type RtGenmsg](#)
[type RtMsg](#)
[type RtNextHop](#)
[type Rusage](#)
[type Signal](#)
 [func \(s Signal\) Signal\(\)](#)
 [func \(s Signal\) String\(\) string](#)
[type SockFilter](#)
 [func LsfJump\(code, k, jt, jf int\) *SockFilter](#)
 [func LsfStmt\(code, k int\) *SockFilter](#)
[type SockFprog](#)
[type Sockaddr](#)
 [func Getpeername\(fd int\) \(sa Sockaddr, err error\)](#)

[func Getsockname\(fd int\) \(sa Sockaddr, err error\)](#)
[type SockaddrInet4](#)
[type SockaddrInet6](#)
[type SockaddrLinklayer](#)
[type SockaddrNetlink](#)
[type SockaddrUnix](#)
[type SocketControlMessage](#)
[type Stat_t](#)
[type Statfs_t](#)
[type SysProcAttr](#)
[type Sysinfo_t](#)
[type Termios](#)
[type Time_t](#)
[func Time\(t *Time_t\) \(tt Time_t, err error\)](#)
[type Timespec](#)
[func NsecToTimespec\(nsec int64\) \(ts Timespec\)](#)
[func \(ts *Timespec\) Nano\(\) int64](#)
[func \(ts *Timespec\) Unix\(\) \(sec int64, nsec int64\)](#)
[type Timeval](#)
[func NsecToTimeval\(nsec int64\) \(tv Timeval\)](#)
[func \(tv *Timeval\) Nano\(\) int64](#)
[func \(tv *Timeval\) Unix\(\) \(sec int64, nsec int64\)](#)
[type Timex](#)
[type Tms](#)
[type Ucred](#)
[func ParseUnixCredentials\(msg *SocketControlMessage\) \(*Ucred, error\)](#)
[type Ustat_t](#)
[type Utimbuf](#)
[type Utsname](#)
[type WaitStatus](#)
[func \(w WaitStatus\) Continued\(\) bool](#)
[func \(w WaitStatus\) CoreDump\(\) bool](#)
[func \(w WaitStatus\) ExitStatus\(\) int](#)
[func \(w WaitStatus\) Exited\(\) bool](#)
[func \(w WaitStatus\) Signal\(\) Signal](#)
[func \(w WaitStatus\) Signaled\(\) bool](#)
[func \(w WaitStatus\) StopSignal\(\) Signal](#)
[func \(w WaitStatus\) Stopped\(\) bool](#)
[func \(w WaitStatus\) TrapCause\(\) int](#)

Package files

[env_unix.go](#) [exec_linux.go](#) [exec_unix.go](#) [lsf_linux.go](#) [netlink_linux.go](#) [sockcmsg_linux.go](#)
[sockcmsg_unix.go](#) [str.go](#) [syscall.go](#) [syscall_linux.go](#) [syscall_linux_amd64.go](#) [syscall_unix.go](#)
[zerrors_linux_amd64.go](#) [zsyscall_linux_amd64.go](#) [zsysnum_linux_amd64.go](#) [ztypes_linux_amd64.go](#)

Constants

```
const (
    AF_ALG           = 0x26
    AF_APPLETALK    = 0x5
    AF_ASH          = 0x12
    AF_ATMPVC       = 0x8
    AF_ATMSVC       = 0x14
    AF_AX25         = 0x3
    AF_BLUETOOTH    = 0x1f
    AF_BRIDGE       = 0x7
    AF_CAIF         = 0x25
    AF_CAN          = 0x1d
    AF_DECnet       = 0xc
    AF_ECONET       = 0x13
    AF_FILE         = 0x1
    AF_IEEE802154  = 0x24
    AF_INET         = 0x2
    AF_INET6        = 0xa
    AF_IPX          = 0x4
    AF_IRDA         = 0x17
    AF_ISDN         = 0x22
    AF_IUCV         = 0x20
    AF_KEY          = 0xf
    AF_LLC          = 0x1a
    AF_LOCAL        = 0x1
    AF_MAX          = 0x27
    AF_NETBEUI     = 0xd
    AF_NETLINK     = 0x10
    AF_NETROM      = 0x6
    AF_PACKET       = 0x11
    AF_PHONET      = 0x23
    AF_PPPOX       = 0x18
    AF_RDS         = 0x15
    AF_ROSE        = 0xb
    AF_ROUTE       = 0x10
    AF_RXRPC       = 0x21
    AF_SECURITY    = 0xe
    AF_SNA         = 0x16
    AF_TIPC        = 0x1e
    AF_UNIX        = 0x1
    AF_UNSPEC      = 0x0
    AF_WANPIPE     = 0x19
    AF_X25         = 0x9
    ARPHRD_ADAPT   = 0x108
    ARPHRD_APPLETLK = 0x8
    ARPHRD_ARCNET  = 0x7
```

ARPHRD_ASH	= 0x30d
ARPHRD_ATM	= 0x13
ARPHRD_AX25	= 0x3
ARPHRD_BIF	= 0x307
ARPHRD_CHAOS	= 0x5
ARPHRD_CISCO	= 0x201
ARPHRD_CSLIP	= 0x101
ARPHRD_CSLIP6	= 0x103
ARPHRD_DDCMP	= 0x205
ARPHRD_DLCI	= 0xf
ARPHRD_ECONET	= 0x30e
ARPHRD_EETHER	= 0x2
ARPHRD_ETHER	= 0x1
ARPHRD_EUI64	= 0x1b
ARPHRD_FCAL	= 0x311
ARPHRD_FCFABRIC	= 0x313
ARPHRD_FCPL	= 0x312
ARPHRD_FCPP	= 0x310
ARPHRD_FDDI	= 0x306
ARPHRD_FRAD	= 0x302
ARPHRD_HDLC	= 0x201
ARPHRD_HIPPI	= 0x30c
ARPHRD_HWX25	= 0x110
ARPHRD_IEEE1394	= 0x18
ARPHRD_IEEE802	= 0x6
ARPHRD_IEEE80211	= 0x321
ARPHRD_IEEE80211_PRISM	= 0x322
ARPHRD_IEEE80211_RADIOTAP	= 0x323
ARPHRD_IEEE802154	= 0x324
ARPHRD_IEEE802154_PHY	= 0x325
ARPHRD_IEEE802_TR	= 0x320
ARPHRD_INFINIBAND	= 0x20
ARPHRD_IPDDP	= 0x309
ARPHRD_IPGRE	= 0x30a
ARPHRD_IRDA	= 0x30f
ARPHRD_LAPB	= 0x204
ARPHRD_LOCALTLK	= 0x305
ARPHRD_LOOPBACK	= 0x304
ARPHRD_METRICOM	= 0x17
ARPHRD_NETROM	= 0x0
ARPHRD_NONE	= 0xffffe
ARPHRD_PIMREG	= 0x30b
ARPHRD_PPP	= 0x200
ARPHRD_PRONET	= 0x4
ARPHRD_RAWHDLC	= 0x206
ARPHRD_ROSE	= 0x10e
ARPHRD_RSRVD	= 0x104
ARPHRD_SIT	= 0x308
ARPHRD_SKIP	= 0x303

ARPHRD_SLIP	= 0x100
ARPHRD_SLIP6	= 0x102
ARPHRD_TUNNEL	= 0x300
ARPHRD_TUNNEL6	= 0x301
ARPHRD_VOID	= 0xffff
ARPHRD_X25	= 0x10f
BPF_A	= 0x10
BPF_ABS	= 0x20
BPF_ADD	= 0x0
BPF_ALU	= 0x4
BPF_AND	= 0x50
BPF_B	= 0x10
BPF_DIV	= 0x30
BPF_H	= 0x8
BPF_IMM	= 0x0
BPF_IND	= 0x40
BPF_JA	= 0x0
BPF_JEQ	= 0x10
BPF_JGE	= 0x30
BPF_JGT	= 0x20
BPF_JMP	= 0x5
BPF_JSET	= 0x40
BPF_K	= 0x0
BPF_LD	= 0x0
BPF_LDX	= 0x1
BPF_LEN	= 0x80
BPF_LSH	= 0x60
BPF_MAJOR_VERSION	= 0x1
BPF_MAXINSNS	= 0x1000
BPF_MEM	= 0x60
BPF_MEMWORDS	= 0x10
BPF_MINOR_VERSION	= 0x1
BPF_MISC	= 0x7
BPF_MSH	= 0xa0
BPF_MUL	= 0x20
BPF_NEG	= 0x80
BPF_OR	= 0x40
BPF_RET	= 0x6
BPF_RSH	= 0x70
BPF_ST	= 0x2
BPF_STX	= 0x3
BPF_SUB	= 0x10
BPF_TAX	= 0x0
BPF_TXA	= 0x80
BPF_W	= 0x0
BPF_X	= 0x8
DT_BLK	= 0x6
DT_CHR	= 0x2
DT_DIR	= 0x4

DT_FIFO	= 0x1
DT_LNK	= 0xa
DT_REG	= 0x8
DT_SOCK	= 0xc
DT_UNKNOWN	= 0x0
DT_WHT	= 0xe
EPOLLERR	= 0x8
EPOLLET	= -0x80000000
EPOLLHUP	= 0x10
EPOLLIN	= 0x1
EPOLLMSG	= 0x400
EPOLLONESHOT	= 0x40000000
EPOLLOUT	= 0x4
EPOLLPRI	= 0x2
EPOLLRDBAND	= 0x80
EPOLLRDHUP	= 0x2000
EPOLLRDNORM	= 0x40
EPOLLWRBAND	= 0x200
EPOLLWRNORM	= 0x100
EPOLL_CLOEXEC	= 0x80000
EPOLL_CTL_ADD	= 0x1
EPOLL_CTL_DEL	= 0x2
EPOLL_CTL_MOD	= 0x3
EPOLL_NONBLOCK	= 0x800
ETH_P_1588	= 0x88f7
ETH_P_8021Q	= 0x8100
ETH_P_802_2	= 0x4
ETH_P_802_3	= 0x1
ETH_P_AARP	= 0x80f3
ETH_P_ALL	= 0x3
ETH_P_AOE	= 0x88a2
ETH_P_ARCNET	= 0x1a
ETH_P_ARP	= 0x806
ETH_P_ATALK	= 0x809b
ETH_P_ATMFATE	= 0x8884
ETH_P_ATMMP0A	= 0x884c
ETH_P_AX25	= 0x2
ETH_P_BPQ	= 0x8ff
ETH_P_CAIF	= 0xf7
ETH_P_CAN	= 0xc
ETH_P_CONTROL	= 0x16
ETH_P_CUST	= 0x6006
ETH_P_DDCMP	= 0x6
ETH_P_DEC	= 0x6000
ETH_P_DIAG	= 0x6005
ETH_P_DNA_DL	= 0x6001
ETH_P_DNA_RC	= 0x6002
ETH_P_DNA_RT	= 0x6003
ETH_P_DSA	= 0x1b

ETH_P_ECONET	= 0x18
ETH_P_EDSA	= 0xdada
ETH_P_FC0E	= 0x8906
ETH_P_FIP	= 0x8914
ETH_P_HDLC	= 0x19
ETH_P_IEEE802154	= 0xf6
ETH_P_IEEEPUP	= 0xa00
ETH_P_IEEEPUPAT	= 0xa01
ETH_P_IP	= 0x800
ETH_P_IPV6	= 0x86dd
ETH_P_IPX	= 0x8137
ETH_P_IRDA	= 0x17
ETH_P_LAT	= 0x6004
ETH_P_LINK_CTL	= 0x886c
ETH_P_LOCALTALK	= 0x9
ETH_P_LOOP	= 0x60
ETH_P_MOBITEX	= 0x15
ETH_P_MPLS_MC	= 0x8848
ETH_P_MPLS_UC	= 0x8847
ETH_P_PAE	= 0x888e
ETH_P_PAUSE	= 0x8808
ETH_P_PHONET	= 0xf5
ETH_P_PPPTALK	= 0x10
ETH_P_PPP_DISC	= 0x8863
ETH_P_PPP_MP	= 0x8
ETH_P_PPP_SES	= 0x8864
ETH_P_PUP	= 0x200
ETH_P_PUPAT	= 0x201
ETH_P_RARP	= 0x8035
ETH_P_SCA	= 0x6007
ETH_P_SLOW	= 0x8809
ETH_P_SNAP	= 0x5
ETH_P_TEB	= 0x6558
ETH_P_TIPC	= 0x88ca
ETH_P_TRAILER	= 0x1c
ETH_P_TR_802_2	= 0x11
ETH_P_WAN_PPP	= 0x7
ETH_P_WCCP	= 0x883e
ETH_P_X25	= 0x805
FD_CLOEXEC	= 0x1
FD_SETSIZE	= 0x400
F_DUPFD	= 0x0
F_DUPFD_CLOEXEC	= 0x406
F_EXLCK	= 0x4
F_GETFD	= 0x1
F_GETFL	= 0x3
F_GETLEASE	= 0x401
F_GETLK	= 0x5
F_GETLK64	= 0x5

F_GETOWN	= 0x9
F_GETOWN_EX	= 0x10
F_GETPIPE_SZ	= 0x408
F_GETSIG	= 0xb
F_LOCK	= 0x1
F_NOTIFY	= 0x402
F_OK	= 0x0
F_RDLCK	= 0x0
F_SETFD	= 0x2
F_SETFL	= 0x4
F_SETLEASE	= 0x400
F_SETLK	= 0x6
F_SETLK64	= 0x6
F_SETLKW	= 0x7
F_SETLKW64	= 0x7
F_SETOWN	= 0x8
F_SETOWN_EX	= 0xf
F_SETPIPE_SZ	= 0x407
F_SETSIG	= 0xa
F_SHLCK	= 0x8
F_TEST	= 0x3
F_TLOCK	= 0x2
F_ULOCK	= 0x0
F_UNLCK	= 0x2
F_WRLCK	= 0x1
IFA_F_DADFAILED	= 0x8
IFA_F_DEPRECATED	= 0x20
IFA_F_HOMEADDRESS	= 0x10
IFA_F_NODAD	= 0x2
IFA_F_OPTIMISTIC	= 0x4
IFA_F_PERMANENT	= 0x80
IFA_F_SECONDARY	= 0x1
IFA_F_TEMPORARY	= 0x1
IFA_F_TENTATIVE	= 0x40
IFA_MAX	= 0x7
IFF_ALLMULTI	= 0x200
IFF_AUTOMEDIA	= 0x4000
IFF_BROADCAST	= 0x2
IFF_DEBUG	= 0x4
IFF_DYNAMIC	= 0x8000
IFF_LOOPBACK	= 0x8
IFF_MASTER	= 0x400
IFF_MULTICAST	= 0x1000
IFF_NOARP	= 0x80
IFF_NOTRAILERS	= 0x20
IFF_NO_PI	= 0x1000
IFF_ONE_QUEUE	= 0x2000
IFF_POINTOPOINT	= 0x10
IFF_PORTSEL	= 0x2000

IFF_PROMISC	= 0x100
IFF_RUNNING	= 0x40
IFF_SLAVE	= 0x800
IFF_TAP	= 0x2
IFF_TUN	= 0x1
IFF_TUN_EXCL	= 0x8000
IFF_UP	= 0x1
IFF_VNET_HDR	= 0x4000
IFNAMSIZ	= 0x10
IN_ACCESS	= 0x1
IN_ALL_EVENTS	= 0xffff
IN_ATTRIB	= 0x4
IN_CLASSA_HOST	= 0xffffffff
IN_CLASSA_MAX	= 0x80
IN_CLASSA_NET	= 0xff000000
IN_CLASSA_NSIFT	= 0x18
IN_CLASSB_HOST	= 0xffff
IN_CLASSB_MAX	= 0x10000
IN_CLASSB_NET	= 0xffff0000
IN_CLASSB_NSIFT	= 0x10
IN_CLASSC_HOST	= 0xff
IN_CLASSC_NET	= 0xffffffff00
IN_CLASSC_NSIFT	= 0x8
IN_CLOEXEC	= 0x80000
IN_CLOSE	= 0x18
IN_CLOSE_NOWRITE	= 0x10
IN_CLOSE_WRITE	= 0x8
IN_CREATE	= 0x100
IN_DELETE	= 0x200
IN_DELETE_SELF	= 0x400
IN_DONT_FOLLOW	= 0x2000000
IN_EXCL_UNLINK	= 0x4000000
IN_IGNORED	= 0x8000
IN_ISDIR	= 0x40000000
IN_LOOPBACKNET	= 0x7f
IN_MASK_ADD	= 0x20000000
IN_MODIFY	= 0x2
IN_MOVE	= 0xc0
IN_MOVED_FROM	= 0x40
IN_MOVED_TO	= 0x80
IN_MOVE_SELF	= 0x800
IN_NONBLOCK	= 0x800
IN_ONESHOT	= 0x80000000
IN_ONLYDIR	= 0x1000000
IN_OPEN	= 0x20
IN_Q_OVERFLOW	= 0x4000
IN_UNMOUNT	= 0x2000
IPPROTO_AH	= 0x33
IPPROTO_COMP	= 0x6c

IPPROTO_DCCP	= 0x21
IPPROTO_DSTOPTS	= 0x3c
IPPROTO_EGP	= 0x8
IPPROTO_ENCAP	= 0x62
IPPROTO_ESP	= 0x32
IPPROTO_FRAGMENT	= 0x2c
IPPROTO_GRE	= 0x2f
IPPROTO_HOPOPTS	= 0x0
IPPROTO_ICMP	= 0x1
IPPROTO_ICMPV6	= 0x3a
IPPROTO_IDP	= 0x16
IPPROTO_IGMP	= 0x2
IPPROTO_IP	= 0x0
IPPROTO_IPIP	= 0x4
IPPROTO_IPV6	= 0x29
IPPROTO_MTP	= 0x5c
IPPROTO_NONE	= 0x3b
IPPROTO_PIM	= 0x67
IPPROTO_PUP	= 0xc
IPPROTO_RAW	= 0xff
IPPROTO_ROUTING	= 0x2b
IPPROTO_RSVP	= 0x2e
IPPROTO_SCTP	= 0x84
IPPROTO_TCP	= 0x6
IPPROTO_TP	= 0x1d
IPPROTO_UDP	= 0x11
IPPROTO_UDPLITE	= 0x88
IPV6_2292DSTOPTS	= 0x4
IPV6_2292HOPLIMIT	= 0x8
IPV6_2292HOPOPTS	= 0x3
IPV6_2292PKTINFO	= 0x2
IPV6_2292PKTOPTIONS	= 0x6
IPV6_2292RTHDR	= 0x5
IPV6_ADDRFORM	= 0x1
IPV6_ADD_MEMBERSHIP	= 0x14
IPV6_AUTHHDR	= 0xa
IPV6_CHECKSUM	= 0x7
IPV6_DROP_MEMBERSHIP	= 0x15
IPV6_DSTOPTS	= 0x3b
IPV6_HOPLIMIT	= 0x34
IPV6_HOPOPTS	= 0x36
IPV6_IPSEC_POLICY	= 0x22
IPV6_JOIN_ANYCAST	= 0x1b
IPV6_JOIN_GROUP	= 0x14
IPV6_LEAVE_ANYCAST	= 0x1c
IPV6_LEAVE_GROUP	= 0x15
IPV6_MTU	= 0x18
IPV6_MTU_DISCOVER	= 0x17
IPV6_MULTICAST_HOPS	= 0x12

IPV6_MULTICAST_IF	= 0x11
IPV6_MULTICAST_LOOP	= 0x13
IPV6_NEXTHOP	= 0x9
IPV6_PKTINFO	= 0x32
IPV6_PMTUDISC_DO	= 0x2
IPV6_PMTUDISC_DONT	= 0x0
IPV6_PMTUDISC_PROBE	= 0x3
IPV6_PMTUDISC_WANT	= 0x1
IPV6_RECVDSTOPTS	= 0x3a
IPV6_RECVERR	= 0x19
IPV6_RECVHOPLIMIT	= 0x33
IPV6_RECVHOPOPTS	= 0x35
IPV6_RECVPKTINFO	= 0x31
IPV6_RECVRTHDR	= 0x38
IPV6_RECVTCLASS	= 0x42
IPV6_ROUTER_ALERT	= 0x16
IPV6_RTHDR	= 0x39
IPV6_RTHDRDSTOPTS	= 0x37
IPV6_RTHDR_LOOSE	= 0x0
IPV6_RTHDR_STRICT	= 0x1
IPV6_RTHDR_TYPE_0	= 0x0
IPV6_RXDSTOPTS	= 0x3b
IPV6_RXHOPOPTS	= 0x36
IPV6_TCLASS	= 0x43
IPV6_UNICAST_HOPS	= 0x10
IPV6_V6ONLY	= 0x1a
IPV6_XFRM_POLICY	= 0x23
IP_ADD_MEMBERSHIP	= 0x23
IP_ADD_SOURCE_MEMBERSHIP	= 0x27
IP_BLOCK_SOURCE	= 0x26
IP_DEFAULT_MULTICAST_LOOP	= 0x1
IP_DEFAULT_MULTICAST_TTL	= 0x1
IP_DF	= 0x4000
IP_DROP_MEMBERSHIP	= 0x24
IP_DROP_SOURCE_MEMBERSHIP	= 0x28
IP_FREEBIND	= 0xf
IP_HDRINCL	= 0x3
IP_IPSEC_POLICY	= 0x10
IP_MAXPACKET	= 0xffff
IP_MAX_MEMBERSHIPS	= 0x14
IP_MF	= 0x2000
IP_MINTTL	= 0x15
IP_MSFILTER	= 0x29
IP_MSS	= 0x240
IP_MTU	= 0xe
IP_MTU_DISCOVER	= 0xa
IP_MULTICAST_IF	= 0x20
IP_MULTICAST_LOOP	= 0x22
IP_MULTICAST_TTL	= 0x21

IP_OFFMASK	= 0x1fff
IP_OPTIONS	= 0x4
IP_ORIGDSTADDR	= 0x14
IP_PASSEC	= 0x12
IP_PKTINFO	= 0x8
IP_PKTOPTIONS	= 0x9
IP_PMTUDISC	= 0xa
IP_PMTUDISC_DO	= 0x2
IP_PMTUDISC_DONT	= 0x0
IP_PMTUDISC_PROBE	= 0x3
IP_PMTUDISC_WANT	= 0x1
IP_RECVERR	= 0xb
IP_RECVOPTS	= 0x6
IP_RECVORIGDSTADDR	= 0x14
IP_RECVRETOPTS	= 0x7
IP_RECVTOS	= 0xd
IP_RECVTTL	= 0xc
IP_RETOPTS	= 0x7
IP_RF	= 0x8000
IP_ROUTER_ALERT	= 0x5
IP_TOS	= 0x1
IP_TRANSPARENT	= 0x13
IP_TTL	= 0x2
IP_UNBLOCK_SOURCE	= 0x25
IP_XFRM_POLICY	= 0x11
LINUX_REBOOT_CMD_CAD_OFF	= 0x0
LINUX_REBOOT_CMD_CAD_ON	= 0x89abcdef
LINUX_REBOOT_CMD_HALT	= 0xcdef0123
LINUX_REBOOT_CMD_KEXEC	= 0x45584543
LINUX_REBOOT_CMD_POWER_OFF	= 0x4321fedc
LINUX_REBOOT_CMD_RESTART	= 0x1234567
LINUX_REBOOT_CMD_RESTART2	= 0xa1b2c3d4
LINUX_REBOOT_CMD_SW_SUSPEND	= 0xd000fcea
LINUX_REBOOT_MAGIC1	= 0xfce1dead
LINUX_REBOOT_MAGIC2	= 0x28121969
LOCK_EX	= 0x2
LOCK_NB	= 0x4
LOCK_SH	= 0x1
LOCK_UN	= 0x8
MADV_DOFORK	= 0xb
MADV_DONTFORK	= 0xa
MADV_DONTNEED	= 0x4
MADV_HUGEPAGE	= 0xe
MADV_HWPOISON	= 0x64
MADV_MERGEABLE	= 0xc
MADV_NOHUGEPAGE	= 0xf
MADV_NORMAL	= 0x0
MADV_RANDOM	= 0x1
MADV_REMOVE	= 0x9

MADV_SEQUENTIAL	= 0x2
MADV_UNMERGEABLE	= 0xd
MADV_WILLNEED	= 0x3
MAP_32BIT	= 0x40
MAP_ANON	= 0x20
MAP_ANONYMOUS	= 0x20
MAP_DENYWRITE	= 0x800
MAP_EXECUTABLE	= 0x1000
MAP_FILE	= 0x0
MAP_FIXED	= 0x10
MAP_GROWSDOWN	= 0x100
MAP_HUGETLB	= 0x40000
MAP_LOCKED	= 0x2000
MAP_NONBLOCK	= 0x10000
MAP_NORESERVE	= 0x4000
MAP_POPULATE	= 0x8000
MAP_PRIVATE	= 0x2
MAP_SHARED	= 0x1
MAP_STACK	= 0x20000
MAP_TYPE	= 0xf
MCL_CURRENT	= 0x1
MCL_FUTURE	= 0x2
MNT_DETACH	= 0x2
MNT_EXPIRE	= 0x4
MNT_FORCE	= 0x1
MSG_CMSG_CLOEXEC	= 0x40000000
MSG_CONFIRM	= 0x800
MSG_CTRUNC	= 0x8
MSG_DONTROUTE	= 0x4
MSG_DONTWAIT	= 0x40
MSG_EOR	= 0x80
MSG_ERRQUEUE	= 0x2000
MSG_FIN	= 0x200
MSG_MORE	= 0x8000
MSG_NOSIGNAL	= 0x4000
MSG_OOB	= 0x1
MSG_PEEK	= 0x2
MSG_PROXY	= 0x10
MSG_RST	= 0x1000
MSG_SYN	= 0x400
MSG_TRUNC	= 0x20
MSG_TRYHARD	= 0x4
MSG_WAITALL	= 0x100
MSG_WAITFORONE	= 0x10000
MS_ACTIVE	= 0x40000000
MS_ASYNC	= 0x1
MS_BIND	= 0x1000
MS_DIRSYNC	= 0x80
MS_INVALIDATE	= 0x2

MS_I_VERSION	= 0x800000
MS_KERNMOUNT	= 0x400000
MS_MANDLOCK	= 0x40
MS_MGC_MSK	= 0xffff0000
MS_MGC_VAL	= 0xc0ed0000
MS_MOVE	= 0x2000
MS_NOATIME	= 0x400
MS_NODEV	= 0x4
MS_NODIRATIME	= 0x800
MS_NOEXEC	= 0x8
MS_NOSUID	= 0x2
MS_NOUSER	= -0x80000000
MS_POSIXACL	= 0x10000
MS_PRIVATE	= 0x40000
MS_RDONLY	= 0x1
MS_REC	= 0x4000
MS_RELATIME	= 0x200000
MS_REMOUNT	= 0x20
MS_RMT_MASK	= 0x800051
MS_SHARED	= 0x100000
MS_SILENT	= 0x8000
MS_SLAVE	= 0x80000
MS_STRICTATIME	= 0x1000000
MS_SYNC	= 0x4
MS_SYNCHRONOUS	= 0x10
MS_UNBINDABLE	= 0x20000
NAME_MAX	= 0xff
NETLINK_ADD_MEMBERSHIP	= 0x1
NETLINK_AUDIT	= 0x9
NETLINK_BROADCAST_ERROR	= 0x4
NETLINK_CONNECTOR	= 0xb
NETLINK_DNRTMSG	= 0xe
NETLINK_DROP_MEMBERSHIP	= 0x2
NETLINK_ECRYPTFS	= 0x13
NETLINK_FIB_LOOKUP	= 0xa
NETLINK_FIREWALL	= 0x3
NETLINK_GENERIC	= 0x10
NETLINK_INET_DIAG	= 0x4
NETLINK_IP6_FW	= 0xd
NETLINK_ISCSI	= 0x8
NETLINK_KOBJECT_UEVENT	= 0xf
NETLINK_NETFILTER	= 0xc
NETLINK_NFLOG	= 0x5
NETLINK_NO_ENOBUFS	= 0x5
NETLINK_PKTINFO	= 0x3
NETLINK_ROUTE	= 0x0
NETLINK_SCSITRANSPORT	= 0x12
NETLINK_SELINUX	= 0x7
NETLINK_UNUSED	= 0x1

NETLINK_USERSOCK	= 0x2
NETLINK_XFRM	= 0x6
NLA_ALIGNTO	= 0x4
NLA_F_NESTED	= 0x8000
NLA_F_NET_BYTEORDER	= 0x4000
NLA_HDRLEN	= 0x4
NLMSG_ALIGNTO	= 0x4
NLMSG_DONE	= 0x3
NLMSG_ERROR	= 0x2
NLMSG_HDRLEN	= 0x10
NLMSG_MIN_TYPE	= 0x10
NLMSG_NOOP	= 0x1
NLMSG_OVERRUN	= 0x4
NLM_F_ACK	= 0x4
NLM_F_APPEND	= 0x800
NLM_F_ATOMIC	= 0x400
NLM_F_CREATE	= 0x400
NLM_F_DUMP	= 0x300
NLM_F_ECHO	= 0x8
NLM_F_EXCL	= 0x200
NLM_F_MATCH	= 0x200
NLM_F_MULTI	= 0x2
NLM_F_REPLACE	= 0x100
NLM_F_REQUEST	= 0x1
NLM_F_ROOT	= 0x100
O_ACCMODE	= 0x3
O_APPEND	= 0x400
O_ASYNC	= 0x2000
O_CLOEXEC	= 0x80000
O_CREAT	= 0x40
O_DIRECT	= 0x4000
O_DIRECTORY	= 0x10000
O_DSYNC	= 0x1000
O_EXCL	= 0x80
O_FSYNC	= 0x101000
O_LARGEFILE	= 0x0
O_NDELAY	= 0x800
O_NOATIME	= 0x40000
O_NOCTTY	= 0x100
O_NOFOLLOW	= 0x20000
O_NONBLOCK	= 0x800
O_RDONLY	= 0x0
O_RDWR	= 0x2
O_RSYNC	= 0x101000
O_SYNC	= 0x101000
O_TRUNC	= 0x200
O_WRONLY	= 0x1
PACKET_ADD_MEMBERSHIP	= 0x1
PACKET_BROADCAST	= 0x1

PACKET_DROP_MEMBERSHIP	= 0x2
PACKET_FASTROUTE	= 0x6
PACKET_HOST	= 0x0
PACKET_LOOPBACK	= 0x5
PACKET_MR_ALLMULTI	= 0x2
PACKET_MR_MULTICAST	= 0x0
PACKET_MR_PROMISC	= 0x1
PACKET_MULTICAST	= 0x2
PACKET_OTHERHOST	= 0x3
PACKET_OUTGOING	= 0x4
PACKET_RECV_OUTPUT	= 0x3
PACKET_RX_RING	= 0x5
PACKET_STATISTICS	= 0x6
PROT_EXEC	= 0x4
PROT_GROWSDOWN	= 0x1000000
PROT_GROWSUP	= 0x2000000
PROT_NONE	= 0x0
PROT_READ	= 0x1
PROT_WRITE	= 0x2
PR_CAPBSET_DROP	= 0x18
PR_CAPBSET_READ	= 0x17
PR_ENDIAN_BIG	= 0x0
PR_ENDIAN_LITTLE	= 0x1
PR_ENDIAN_PPC_LITTLE	= 0x2
PR_FPEMU_NOPRINT	= 0x1
PR_FPEMU_SIGFPE	= 0x2
PR_FP_EXC_ASYNC	= 0x2
PR_FP_EXC_DISABLED	= 0x0
PR_FP_EXC_DIV	= 0x10000
PR_FP_EXC_INV	= 0x100000
PR_FP_EXC_NONRECOV	= 0x1
PR_FP_EXC_OVF	= 0x20000
PR_FP_EXC_PRECISE	= 0x3
PR_FP_EXC_RES	= 0x80000
PR_FP_EXC_SW_ENABLE	= 0x80
PR_FP_EXC_UND	= 0x40000
PR_GET_DUMPABLE	= 0x3
PR_GET_ENDIAN	= 0x13
PR_GET_FPEMU	= 0x9
PR_GET_FPEXC	= 0xb
PR_GET_KEEPCAPS	= 0x7
PR_GET_NAME	= 0x10
PR_GET_PDEATHSIG	= 0x2
PR_GET_SECCOMP	= 0x15
PR_GET_SECUREBITS	= 0x1b
PR_GET_TIMERSLACK	= 0x1e
PR_GET_TIMING	= 0xd
PR_GET_TSC	= 0x19
PR_GET_UNALIGN	= 0x5

PR_MCE_KILL	= 0x21
PR_MCE_KILL_CLEAR	= 0x0
PR_MCE_KILL_DEFAULT	= 0x2
PR_MCE_KILL_EARLY	= 0x1
PR_MCE_KILL_GET	= 0x22
PR_MCE_KILL_LATE	= 0x0
PR_MCE_KILL_SET	= 0x1
PR_SET_DUMPABLE	= 0x4
PR_SET_ENDIAN	= 0x14
PR_SET_FPEMU	= 0xa
PR_SET_FPEXC	= 0xc
PR_SET_KEEPCAPS	= 0x8
PR_SET_NAME	= 0xf
PR_SET_PDEATHSIG	= 0x1
PR_SET_PTRACER	= 0x59616d61
PR_SET_SECCOMP	= 0x16
PR_SET_SECUREBITS	= 0x1c
PR_SET_TIMERSLACK	= 0x1d
PR_SET_TIMING	= 0xe
PR_SET_TSC	= 0x1a
PR_SET_UNALIGN	= 0x6
PR_TASK_PERF_EVENTS_DISABLE	= 0x1f
PR_TASK_PERF_EVENTS_ENABLE	= 0x20
PR_TIMING_STATISTICAL	= 0x0
PR_TIMING_TIMESTAMP	= 0x1
PR_TSC_ENABLE	= 0x1
PR_TSC_SIGSEGV	= 0x2
PR_UNALIGN_NOPRINT	= 0x1
PR_UNALIGN_SIGBUS	= 0x2
PTRACE_ARCH_PRCTL	= 0x1e
PTRACE_ATTACH	= 0x10
PTRACE_CONT	= 0x7
PTRACE_DETACH	= 0x11
PTRACE_EVENT_CLONE	= 0x3
PTRACE_EVENT_EXEC	= 0x4
PTRACE_EVENT_EXIT	= 0x6
PTRACE_EVENT_FORK	= 0x1
PTRACE_EVENT_VFORK	= 0x2
PTRACE_EVENT_VFORK_DONE	= 0x5
PTRACE_GETEVENTMSG	= 0x4201
PTRACE_GETFPREGS	= 0xe
PTRACE_GETFPXREGS	= 0x12
PTRACE_GETREGS	= 0xc
PTRACE_GETREGSET	= 0x4204
PTRACE_GETSIGINFO	= 0x4202
PTRACE_GET_THREAD_AREA	= 0x19
PTRACE_KILL	= 0x8
PTRACE_OLDSETOPTIONS	= 0x15
PTRACE_O_MASK	= 0x7f

PTRACE_0_TRACECLONE	= 0x8
PTRACE_0_TRACEEXEC	= 0x10
PTRACE_0_TRACEEXIT	= 0x40
PTRACE_0_TRACEFORK	= 0x2
PTRACE_0_TRACESYSGOOD	= 0x1
PTRACE_0_TRACEVFORK	= 0x4
PTRACE_0_TRACEVFORKDONE	= 0x20
PTRACE_PEEKDATA	= 0x2
PTRACE_PEEKTEXT	= 0x1
PTRACE_PEEKUSR	= 0x3
PTRACE_POKEDATA	= 0x5
PTRACE_POKETEXT	= 0x4
PTRACE_POKEUSR	= 0x6
PTRACE_SETFPREGS	= 0xf
PTRACE_SETFPXREGS	= 0x13
PTRACE_SETOPTIONS	= 0x4200
PTRACE_SETREGS	= 0xd
PTRACE_SETREGSET	= 0x4205
PTRACE_SETSIGINFO	= 0x4203
PTRACE_SET_THREAD_AREA	= 0x1a
PTRACE_SINGLEBLOCK	= 0x21
PTRACE_SINGLESTEP	= 0x9
PTRACE_SYSCALL	= 0x18
PTRACE_SYSEMU	= 0x1f
PTRACE_SYSEMU_SINGLESTEP	= 0x20
PTRACE_TRACEME	= 0x0
RLIMIT_AS	= 0x9
RLIMIT_CORE	= 0x4
RLIMIT_CPU	= 0x0
RLIMIT_DATA	= 0x2
RLIMIT_FSIZE	= 0x1
RLIMIT_NOFILE	= 0x7
RLIMIT_STACK	= 0x3
RLIM_INFINITY	= -0x1
RTAX_ADVMS	= 0x8
RTAX_CWND	= 0x7
RTAX_FEATURES	= 0xc
RTAX_FEATURE_ALLFRAG	= 0x8
RTAX_FEATURE_ECN	= 0x1
RTAX_FEATURE_SACK	= 0x2
RTAX_FEATURE_TIMESTAMP	= 0x4
RTAX_HOPLIMIT	= 0xa
RTAX_INITCWND	= 0xb
RTAX_INITRWND	= 0xe
RTAX_LOCK	= 0x1
RTAX_MAX	= 0xe
RTAX_MTU	= 0x2
RTAX_REORDERING	= 0x9
RTAX_RTO_MIN	= 0xd

RTAX_RTT	= 0x4
RTAX_RTTVAR	= 0x5
RTAX_SSTHRESH	= 0x6
RTAX_UNSPEC	= 0x0
RTAX_WINDOW	= 0x3
RTA_ALIGNTO	= 0x4
RTA_MAX	= 0x10
RTCF_DIRECTSRC	= 0x4000000
RTCF_DOREDIRECT	= 0x1000000
RTCF_LOG	= 0x2000000
RTCF_MASQ	= 0x400000
RTCF_NAT	= 0x800000
RTCF_VALVE	= 0x200000
RTF_ADDRCLASSMASK	= 0xf8000000
RTF_ADDRCONF	= 0x40000
RTF_ALLONLINK	= 0x20000
RTF_BROADCAST	= 0x10000000
RTF_CACHE	= 0x1000000
RTF_DEFAULT	= 0x10000
RTF_DYNAMIC	= 0x10
RTF_FLOW	= 0x2000000
RTF_GATEWAY	= 0x2
RTF_HOST	= 0x4
RTF_INTERFACE	= 0x40000000
RTF_IRTT	= 0x100
RTF_LINKRT	= 0x100000
RTF_LOCAL	= 0x80000000
RTF_MODIFIED	= 0x20
RTF_MSS	= 0x40
RTF_MTU	= 0x40
RTF_MULTICAST	= 0x20000000
RTF_NAT	= 0x8000000
RTF_NOFORWARD	= 0x1000
RTF_NONEXTHOP	= 0x200000
RTF_NOPMTUDISC	= 0x4000
RTF_POLICY	= 0x4000000
RTF_REINSTATE	= 0x8
RTF_REJECT	= 0x200
RTF_STATIC	= 0x400
RTF_THROW	= 0x2000
RTF_UP	= 0x1
RTF_WINDOW	= 0x80
RTF_XRESOLVE	= 0x800
RTM_BASE	= 0x10
RTM_DELACTION	= 0x31
RTM_DELADDR	= 0x15
RTM_DELADRLABEL	= 0x49
RTM_DELLINK	= 0x11
RTM_DELNEIGH	= 0x1d

RTM_DELQDISC	= 0x25
RTM_DELROUTE	= 0x19
RTM_DELRULE	= 0x21
RTM_DELTCLASS	= 0x29
RTM_DELTFILTER	= 0x2d
RTM_F_CLONED	= 0x200
RTM_F_EQUALIZE	= 0x400
RTM_F_NOTIFY	= 0x100
RTM_F_PREFIX	= 0x800
RTM_GETACTION	= 0x32
RTM_GETADDR	= 0x16
RTM_GETADDRLABEL	= 0x4a
RTM_GETANYCAST	= 0x3e
RTM_GETDCB	= 0x4e
RTM_GETLINK	= 0x12
RTM_GETMULTICAST	= 0x3a
RTM_GETNEIGH	= 0x1e
RTM_GETNEIGHTBL	= 0x42
RTM_GETQDISC	= 0x26
RTM_GETROUTE	= 0x1a
RTM_GETRULE	= 0x22
RTM_GETTCLASS	= 0x2a
RTM_GETTFILTER	= 0x2e
RTM_MAX	= 0x4f
RTM_NEWACTION	= 0x30
RTM_NEWADDR	= 0x14
RTM_NEWADDRLABEL	= 0x48
RTM_NEWLINK	= 0x10
RTM_NEWNDUSEROPT	= 0x44
RTM_NEWNEIGH	= 0x1c
RTM_NEWNEIGHTBL	= 0x40
RTM_NEWPREFIX	= 0x34
RTM_NEWQDISC	= 0x24
RTM_NEWROUTE	= 0x18
RTM_NEWRULE	= 0x20
RTM_NEWTCLASS	= 0x28
RTM_NEWTFILTER	= 0x2c
RTM_NR_FAMILIES	= 0x10
RTM_NR_MSGTYPES	= 0x40
RTM_SETDCB	= 0x4f
RTM_SETLINK	= 0x13
RTM_SETNEIGHTBL	= 0x43
RTNH_ALIGNTO	= 0x4
RTNH_F_DEAD	= 0x1
RTNH_F_ONLINK	= 0x4
RTNH_F_PERVASIVE	= 0x2
RTN_MAX	= 0xb
RTPROT_BIRD	= 0xc
RTPROT_BOOT	= 0x3

RTPROT_DHCP	= 0x10
RTPROT_DNRROUTED	= 0xd
RTPROT_GATED	= 0x8
RTPROT_KERNEL	= 0x2
RTPROT_MRT	= 0xa
RTPROT_NTK	= 0xf
RTPROT_RA	= 0x9
RTPROT_REDIRECT	= 0x1
RTPROT_STATIC	= 0x4
RTPROT_UNSPEC	= 0x0
RTPROT_XORP	= 0xe
RTPROT_ZEBRA	= 0xb
RT_CLASS_DEFAULT	= 0xfd
RT_CLASS_LOCAL	= 0xff
RT_CLASS_MAIN	= 0xfe
RT_CLASS_MAX	= 0xff
RT_CLASS_UNSPEC	= 0x0
RUSAGE_CHILDREN	= -0x1
RUSAGE_SELF	= 0x0
RUSAGE_THREAD	= 0x1
SCM_CREDENTIALS	= 0x2
SCM_RIGHTS	= 0x1
SCM_TIMESTAMP	= 0x1d
SCM_TIMESTAMPING	= 0x25
SCM_TIMESTAMPNS	= 0x23
SHUT_RD	= 0x0
SHUT_RDWR	= 0x2
SHUT_WR	= 0x1
SIOCADDLCI	= 0x8980
SIOCADDMULTI	= 0x8931
SIOCADDRT	= 0x890b
SIOCATMARK	= 0x8905
SIOCДАРP	= 0x8953
SIOCDELDLCI	= 0x8981
SIOCDELMULTI	= 0x8932
SIOCDELRT	= 0x890c
SIOCDEVPRIVATE	= 0x89f0
SIOCДИFADDR	= 0x8936
SIOCDRARP	= 0x8960
SIOCGARP	= 0x8954
SIOCGIFADDR	= 0x8915
SIOCGIFBR	= 0x8940
SIOCGIFBRDADDR	= 0x8919
SIOCGIFCONF	= 0x8912
SIOCGIFCOUNT	= 0x8938
SIOCGIFDSTADDR	= 0x8917
SIOCGIFENCAP	= 0x8925
SIOCGIFFLAGS	= 0x8913
SIOCGIFHWADDR	= 0x8927

SIOCGIFINDEX	= 0x8933
SIOCGIFMAP	= 0x8970
SIOCGIFMEM	= 0x891f
SIOCGIFMETRIC	= 0x891d
SIOCGIFMTU	= 0x8921
SIOCGIFNAME	= 0x8910
SIOCGIFNETMASK	= 0x891b
SIOCGIFPFLAGS	= 0x8935
SIOCGIFSLAVE	= 0x8929
SIOCGIFTXQLEN	= 0x8942
SIOCGPGRP	= 0x8904
SIOCGRARP	= 0x8961
SIOCGSTAMP	= 0x8906
SIOCGSTAMPNS	= 0x8907
SIOCPROTOPRIVATE	= 0x89e0
SIOCRTMSG	= 0x890d
SIOCSARP	= 0x8955
SIOCSIFADDR	= 0x8916
SIOCSIFBR	= 0x8941
SIOCSIFBRDADDR	= 0x891a
SIOCSIFDSTADDR	= 0x8918
SIOCSIFENCAP	= 0x8926
SIOCSIFFLAGS	= 0x8914
SIOCSIFHWADDR	= 0x8924
SIOCSIFHWBROADCAST	= 0x8937
SIOCSIFLINK	= 0x8911
SIOCSIFMAP	= 0x8971
SIOCSIFMEM	= 0x8920
SIOCSIFMETRIC	= 0x891e
SIOCSIFMTU	= 0x8922
SIOCSIFNAME	= 0x8923
SIOCSIFNETMASK	= 0x891c
SIOCSIFPFLAGS	= 0x8934
SIOCSIFSLAVE	= 0x8930
SIOCSIFTXQLEN	= 0x8943
SIOCSPGRP	= 0x8902
SIOCSRARP	= 0x8962
SOCK_CLOEXEC	= 0x80000
SOCK_DCCP	= 0x6
SOCK_DGRAM	= 0x2
SOCK_NONBLOCK	= 0x800
SOCK_PACKET	= 0xa
SOCK_RAW	= 0x3
SOCK_RDM	= 0x4
SOCK_SEQPACKET	= 0x5
SOCK_STREAM	= 0x1
SOL_AAL	= 0x109
SOL_ATM	= 0x108
SOL_DECNET	= 0x105

SOL_ICMPV6	= 0x3a
SOL_IP	= 0x0
SOL_IPV6	= 0x29
SOL_IRDA	= 0x10a
SOL_PACKET	= 0x107
SOL_RAW	= 0xff
SOL_SOCKET	= 0x1
SOL_TCP	= 0x6
SOL_X25	= 0x106
SOMAXCONN	= 0x80
SO_ACCEPTCONN	= 0x1e
SO_ATTACH_FILTER	= 0x1a
SO_BINDTODEVICE	= 0x19
SO_BROADCAST	= 0x6
SO_BSDCOMPAT	= 0xe
SO_DEBUG	= 0x1
SO_DETACH_FILTER	= 0x1b
SO_DOMAIN	= 0x27
SO_DONTROUTE	= 0x5
SO_ERROR	= 0x4
SO_KEEPALIVE	= 0x9
SO_LINGER	= 0xd
SO_MARK	= 0x24
SO_NO_CHECK	= 0xb
SO_OOBINLINE	= 0xa
SO_PASSCRED	= 0x10
SO_PASSSEC	= 0x22
SO_PEERCREC	= 0x11
SO_PEERNAME	= 0x1c
SO_PEERSEC	= 0x1f
SO_PRIORITY	= 0xc
SO_PROTOCOL	= 0x26
SO_RCVBUF	= 0x8
SO_RCVBUFFORCE	= 0x21
SO_RCVLOWAT	= 0x12
SO_RCVTIMEO	= 0x14
SO_REUSEADDR	= 0x2
SO_RXQ_OVFL	= 0x28
SO_SECURITY_AUTHENTICATION	= 0x16
SO_SECURITY_ENCRYPTION_NETWORK	= 0x18
SO_SECURITY_ENCRYPTION_TRANSPORT	= 0x17
SO_SNDBUF	= 0x7
SO_SNDBUFFORCE	= 0x20
SO_SNDLOWAT	= 0x13
SO_SNDTIMEO	= 0x15
SO_TIMESTAMP	= 0x1d
SO_TIMESTAMPING	= 0x25
SO_TIMESTAMPNS	= 0x23
SO_TYPE	= 0x3

S_BLKSIZE	= 0x200
S_IEXEC	= 0x40
S_IFBLK	= 0x6000
S_IFCHR	= 0x2000
S_IFDIR	= 0x4000
S_IFIFO	= 0x1000
S_IFLNK	= 0xa000
S_IFMT	= 0xf000
S_IFREG	= 0x8000
S_IFSOCK	= 0xc000
S_IREAD	= 0x100
S_IRGRP	= 0x20
S_IROTH	= 0x4
S_IRUSR	= 0x100
S_IRWXG	= 0x38
S_IRWXO	= 0x7
S_IRWXU	= 0x1c0
S_ISGID	= 0x400
S_ISUID	= 0x800
S_ISVTX	= 0x200
S_IWGRP	= 0x10
S_IWOTH	= 0x2
S_IWRITE	= 0x80
S_IWUSR	= 0x80
S_IXGRP	= 0x8
S_IXOTH	= 0x1
S_IXUSR	= 0x40
TCP_CONGESTION	= 0xd
TCP_CORK	= 0x3
TCP_DEFER_ACCEPT	= 0x9
TCP_INFO	= 0xb
TCP_KEEPCNT	= 0x6
TCP_KEEPIDLE	= 0x4
TCP_KEEPINTVL	= 0x5
TCP_LINGER2	= 0x8
TCP_MAXSEG	= 0x2
TCP_MAXWIN	= 0xffff
TCP_MAX_WINSHIFT	= 0xe
TCP_MD5SIG	= 0xe
TCP_MD5SIG_MAXKEYLEN	= 0x50
TCP_MSS	= 0x200
TCP_NODELAY	= 0x1
TCP_QUICKACK	= 0xc
TCP_SYNCNT	= 0x7
TCP_WINDOW_CLAMP	= 0xa
TIOCCBRK	= 0x5428
TIOCCONS	= 0x541d
TIOCEXCL	= 0x540c
TIOCGDEV	= 0x80045432

TIOCGETD	= 0x5424
TIOCGICOUNT	= 0x545d
TIOCGLCKTRMIO\$	= 0x5456
TIOCGPGRP	= 0x540f
TIOCGPTN	= 0x80045430
TIOCGRS485	= 0x542e
TIOCGSERIAL	= 0x541e
TIOCGSID	= 0x5429
TIOCGSOFTCAR	= 0x5419
TIOCGWINSZ	= 0x5413
TIOCINQ	= 0x541b
TIOCLINUX	= 0x541c
TIOCMBIC	= 0x5417
TIOCMBIS	= 0x5416
TIOCMGET	= 0x5415
TIOCMWAIT	= 0x545c
TIOCMSET	= 0x5418
TIOCM_CAR	= 0x40
TIOCM_CD	= 0x40
TIOCM_CTS	= 0x20
TIOCM_DSR	= 0x100
TIOCM_DTR	= 0x2
TIOCM_LE	= 0x1
TIOCM_RI	= 0x80
TIOCM_RNG	= 0x80
TIOCM_RTS	= 0x4
TIOCM_SR	= 0x10
TIOCM_ST	= 0x8
TIOCNOTTY	= 0x5422
TIOCNXCL	= 0x540d
TIOCOUTQ	= 0x5411
TIOCPKT	= 0x5420
TIOCPKT_DATA	= 0x0
TIOCPKT_DOSTOP	= 0x20
TIOCPKT_FLUSHREAD	= 0x1
TIOCPKT_FLUSHWRITE	= 0x2
TIOCPKT_IOCTL	= 0x40
TIOCPKT_NOSTOP	= 0x10
TIOCPKT_START	= 0x8
TIOCPKT_STOP	= 0x4
TIOCSBRK	= 0x5427
TIOCSCTTY	= 0x540e
TIOCSERCONFIG	= 0x5453
TIOCSERGETLSR	= 0x5459
TIOCSERGETMULTI	= 0x545a
TIOCSERGSTRUCT	= 0x5458
TIOCSERGWILD	= 0x5454
TIOCSERSETMULTI	= 0x545b
TIOCSERSWILD	= 0x5455

TIOCSER_TEMT	= 0x1
TIOCSETD	= 0x5423
TIOCSIG	= 0x40045436
TIOCSLCKTRMIOS	= 0x5457
TIOCSPGRP	= 0x5410
TIOCSPTLCK	= 0x40045431
TIOCSRS485	= 0x542f
TIOCSSERIAL	= 0x541f
TIOCSSOFTCAR	= 0x541a
TIOCSTI	= 0x5412
TIOCSWINSZ	= 0x5414
TUNATTACHFILTER	= 0x401054d5
TUNDETACHFILTER	= 0x401054d6
TUNGETFEATURES	= 0x800454cf
TUNGETIFF	= 0x800454d2
TUNGETSNDBUF	= 0x800454d3
TUNGETVNETHDRSZ	= 0x800454d7
TUNSETDEBUG	= 0x400454c9
TUNSETGROUP	= 0x400454ce
TUNSETIFF	= 0x400454ca
TUNSETLINK	= 0x400454cd
TUNSETNOCSUM	= 0x400454c8
TUNSETOFFLOAD	= 0x400454d0
TUNSETOWNER	= 0x400454cc
TUNSETPERSIST	= 0x400454cb
TUNSETSNDBUF	= 0x400454d4
TUNSETTXFILTER	= 0x400454d1
TUNSETVNETHDRSZ	= 0x400454d8
WALL	= 0x40000000
WCLONE	= 0x80000000
WCONTINUED	= 0x8
WEXITED	= 0x4
WNOHANG	= 0x1
WNOTHREAD	= 0x20000000
WNOWAIT	= 0x1000000
WORDSIZE	= 0x40
WSTOPPED	= 0x2
WUNTRACED	= 0x2

)

```
const (
    E2BIG          = Errno(0x7)
    EACCES        = Errno(0xd)
    EADDRINUSE    = Errno(0x62)
    EADDRNOTAVAIL = Errno(0x63)
    EADV          = Errno(0x44)
    EAFNOSUPPORT  = Errno(0x61)
    EAGAIN        = Errno(0xb)
    EALREADY      = Errno(0x72)
    EBADE         = Errno(0x34)
```

EBADF	= Errno(0x9)
EBADFD	= Errno(0x4d)
EBADMSG	= Errno(0x4a)
EBADR	= Errno(0x35)
EBADRQC	= Errno(0x38)
EBADSLT	= Errno(0x39)
EBFONT	= Errno(0x3b)
EBUSY	= Errno(0x10)
ECANCELED	= Errno(0x7d)
ECHILD	= Errno(0xa)
ECHRNG	= Errno(0x2c)
ECOMM	= Errno(0x46)
ECONNABORTED	= Errno(0x67)
ECONNREFUSED	= Errno(0x6f)
ECONNRESET	= Errno(0x68)
EDEADLK	= Errno(0x23)
EDEADLOCK	= Errno(0x23)
EDESTADDRREQ	= Errno(0x59)
EDOM	= Errno(0x21)
EDOTDOT	= Errno(0x49)
EDQUOT	= Errno(0x7a)
EEXIST	= Errno(0x11)
EFAULT	= Errno(0xe)
EFBIG	= Errno(0x1b)
EHOSTDOWN	= Errno(0x70)
EHOSTUNREACH	= Errno(0x71)
EIDRM	= Errno(0x2b)
EILSEQ	= Errno(0x54)
EINPROGRESS	= Errno(0x73)
EINTR	= Errno(0x4)
EINVAL	= Errno(0x16)
EIO	= Errno(0x5)
EISCONN	= Errno(0x6a)
EISDIR	= Errno(0x15)
EISNAM	= Errno(0x78)
EKEYEXPIRED	= Errno(0x7f)
EKEYREJECTED	= Errno(0x81)
EKEYREVOKED	= Errno(0x80)
EL2HLT	= Errno(0x33)
EL2NSYNC	= Errno(0x2d)
EL3HLT	= Errno(0x2e)
EL3RST	= Errno(0x2f)
ELIBACC	= Errno(0x4f)
ELIBBAD	= Errno(0x50)
ELIBEXEC	= Errno(0x53)
ELIBMAX	= Errno(0x52)
ELIBSCN	= Errno(0x51)
ELNRNG	= Errno(0x30)
ELOOP	= Errno(0x28)
EMEDIUMTYPE	= Errno(0x7c)

EMFILE	= Errno(0x18)
EMLINK	= Errno(0x1f)
EMSGSIZE	= Errno(0x5a)
EMULTIHOP	= Errno(0x48)
ENAMETOOLONG	= Errno(0x24)
ENAVAIL	= Errno(0x77)
ENETDOWN	= Errno(0x64)
ENETRESET	= Errno(0x66)
ENETUNREACH	= Errno(0x65)
ENFILE	= Errno(0x17)
ENOANO	= Errno(0x37)
ENOBUFS	= Errno(0x69)
ENOCSI	= Errno(0x32)
ENODATA	= Errno(0x3d)
ENODEV	= Errno(0x13)
ENOENT	= Errno(0x2)
ENOEXEC	= Errno(0x8)
ENOKEY	= Errno(0x7e)
ENOLCK	= Errno(0x25)
ENOLINK	= Errno(0x43)
ENOMEDIUM	= Errno(0x7b)
ENOMEM	= Errno(0xc)
ENOMSG	= Errno(0x2a)
ENONET	= Errno(0x40)
ENOPKG	= Errno(0x41)
ENOPROTOOPT	= Errno(0x5c)
ENOSPC	= Errno(0x1c)
ENOSR	= Errno(0x3f)
ENOSTR	= Errno(0x3c)
ENOSYS	= Errno(0x26)
ENOTBLK	= Errno(0xf)
ENOTCONN	= Errno(0x6b)
ENOTDIR	= Errno(0x14)
ENOTEMPTY	= Errno(0x27)
ENOTNAM	= Errno(0x76)
ENOTRECOVERABLE	= Errno(0x83)
ENOTSOCK	= Errno(0x58)
ENOTSUP	= Errno(0x5f)
ENOTTY	= Errno(0x19)
ENOTUNIQ	= Errno(0x4c)
ENXIO	= Errno(0x6)
EOPNOTSUPP	= Errno(0x5f)
E_OVERFLOW	= Errno(0x4b)
EOWNERDEAD	= Errno(0x82)
EPERM	= Errno(0x1)
EPFNOSUPPORT	= Errno(0x60)
EPIPE	= Errno(0x20)
EPROTO	= Errno(0x47)
EPROTONOSUPPORT	= Errno(0x5d)
EPROTOTYPE	= Errno(0x5b)

```
ERANGE          = Errno(0x22)
EREMCHG         = Errno(0x4e)
EREMOTE         = Errno(0x42)
EREMOTEIO       = Errno(0x79)
ERESTART        = Errno(0x55)
ERFKILL         = Errno(0x84)
EROFS           = Errno(0x1e)
ESHUTDOWN       = Errno(0x6c)
ESOCKTNOSUPPORT = Errno(0x5e)
ESPIPE          = Errno(0x1d)
ESRCH           = Errno(0x3)
ESRMNT          = Errno(0x45)
ESTALE          = Errno(0x74)
ESTRPIPE        = Errno(0x56)
ETIME           = Errno(0x3e)
ETIMEDOUT       = Errno(0x6e)
ETOOMANYREFS    = Errno(0x6d)
ETXTBSY         = Errno(0x1a)
EUCLEAN         = Errno(0x75)
EUNATCH         = Errno(0x31)
EUSERS          = Errno(0x57)
EWOULDBLOCK     = Errno(0xb)
EXDEV           = Errno(0x12)
EXFULL          = Errno(0x36)
```

)

Errors

```
const (
    SIGABRT = Signal(0x6)
    SIGALRM = Signal(0xe)
    SIGBUS  = Signal(0x7)
    SIGCHLD = Signal(0x11)
    SIGCLD  = Signal(0x11)
    SIGCONT = Signal(0x12)
    SIGFPE  = Signal(0x8)
    SIGHUP  = Signal(0x1)
    SIGILL  = Signal(0x4)
    SIGINT  = Signal(0x2)
    SIGIO   = Signal(0x1d)
    SIGIOT  = Signal(0x6)
    SIGKILL = Signal(0x9)
    SIGPIPE = Signal(0xd)
    SIGPOLL = Signal(0x1d)
    SIGPROF = Signal(0x1b)
    SIGPWR  = Signal(0x1e)
    SIGQUIT = Signal(0x3)
    SIGSEGV = Signal(0xb)
    SIGSTKFLT = Signal(0x10)
```

```
SIGSTOP    = Signal(0x13)
SIGSYS     = Signal(0x1f)
SIGTERM    = Signal(0xf)
SIGTRAP    = Signal(0x5)
SIGTSTP    = Signal(0x14)
SIGTTIN    = Signal(0x15)
SIGTTOU    = Signal(0x16)
SIGUNUSED  = Signal(0x1f)
SIGURG     = Signal(0x17)
SIGUSR1    = Signal(0xa)
SIGUSR2    = Signal(0xc)
SIGVTALRM  = Signal(0x1a)
SIGWINCH   = Signal(0x1c)
SIGXCPU    = Signal(0x18)
SIGXFSZ    = Signal(0x19)
)
```

Signals

```
const (
    SYS_READ           = 0
    SYS_WRITE          = 1
    SYS_OPEN           = 2
    SYS_CLOSE          = 3
    SYS_STAT           = 4
    SYS_FSTAT          = 5
    SYS_LSTAT          = 6
    SYS_POLL           = 7
    SYS_LSEEK          = 8
    SYS_MMAP           = 9
    SYS_MPROTECT       = 10
    SYS_MUNMAP         = 11
    SYS_BRK            = 12
    SYS_RT_SIGACTION   = 13
    SYS_RT_SIGPROCMASK = 14
    SYS_RT_SIGRETURN   = 15
    SYS_IOCTL          = 16
    SYS_PREAD64        = 17
    SYS_PWRITE64       = 18
    SYS_READV          = 19
    SYS_WRITEV         = 20
    SYS_ACCESS         = 21
    SYS_PIPE           = 22
    SYS_SELECT         = 23
    SYS_SCHED_YIELD    = 24
    SYS_MREMAP         = 25
    SYS_MSYNC          = 26
    SYS_MINCORE        = 27
    SYS_MADVISE        = 28
)
```

SYS_SHMGET	= 29
SYS_SHMAT	= 30
SYS_SHMCTL	= 31
SYS_DUP	= 32
SYS_DUP2	= 33
SYS_PAUSE	= 34
SYS_NANOSLEEP	= 35
SYS_GETITIMER	= 36
SYS_ALARM	= 37
SYS_SETITIMER	= 38
SYS_GETPID	= 39
SYS_SENDFILE	= 40
SYS_SOCKET	= 41
SYS_CONNECT	= 42
SYS_ACCEPT	= 43
SYS_SENDTO	= 44
SYS_RECVFROM	= 45
SYS_SENDMSG	= 46
SYS_RECVMSG	= 47
SYS_SHUTDOWN	= 48
SYS_BIND	= 49
SYS_LISTEN	= 50
SYS_GETSOCKNAME	= 51
SYS_GETPEERNAME	= 52
SYS_SOCKETPAIR	= 53
SYS_SETSOCKOPT	= 54
SYS_GETSOCKOPT	= 55
SYS_CLONE	= 56
SYS_FORK	= 57
SYS_VFORK	= 58
SYS_EXECVE	= 59
SYS_EXIT	= 60
SYS_WAIT4	= 61
SYS_KILL	= 62
SYS_UNAME	= 63
SYS_SEMGET	= 64
SYS_SEMOP	= 65
SYS_SEMCTL	= 66
SYS_SHMDT	= 67
SYS_MSGGET	= 68
SYS_MSGSND	= 69
SYS_MSGRCV	= 70
SYS_MSGCTL	= 71
SYS_FCNTL	= 72
SYS_FLOCK	= 73
SYS_FSYNC	= 74
SYS_FDATASYNC	= 75
SYS_TRUNCATE	= 76
SYS_FTRUNCATE	= 77
SYS_GETDENTS	= 78

SYS_GETCWD	= 79
SYS_CHDIR	= 80
SYS_FCHDIR	= 81
SYS_RENAME	= 82
SYS_MKDIR	= 83
SYS_RMDIR	= 84
SYS_CREAT	= 85
SYS_LINK	= 86
SYS_UNLINK	= 87
SYS_SYMLINK	= 88
SYS_READLINK	= 89
SYS_CHMOD	= 90
SYS_FCHMOD	= 91
SYS_CHOWN	= 92
SYS_FCHOWN	= 93
SYS_LCHOWN	= 94
SYS_UMASK	= 95
SYS_GETTIMEOFDAY	= 96
SYS_GETRLIMIT	= 97
SYS_GETRUSAGE	= 98
SYS_SYSINFO	= 99
SYS_TIMES	= 100
SYS_PTRACE	= 101
SYS_GETUID	= 102
SYS_SYSLOG	= 103
SYS_GETGID	= 104
SYS_SETUID	= 105
SYS_SETGID	= 106
SYS_GETEUID	= 107
SYS_GETEGID	= 108
SYS_SETPGID	= 109
SYS_GETPPID	= 110
SYS_GETPGRP	= 111
SYS_SETSID	= 112
SYS_SETREUID	= 113
SYS_SETREGID	= 114
SYS_GETGROUPS	= 115
SYS_SETGROUPS	= 116
SYS_SETRESUID	= 117
SYS_GETRESUID	= 118
SYS_SETRESGID	= 119
SYS_GETRESGID	= 120
SYS_GETPGID	= 121
SYS_SETFSUID	= 122
SYS_SETFSGID	= 123
SYS_GETSID	= 124
SYS_CAPGET	= 125
SYS_CAPSET	= 126
SYS_RT_SIGPENDING	= 127
SYS_RT_SIGTIMEDWAIT	= 128

SYS_RT_SIGQUEUEINFO	= 129
SYS_RT_SIGSUSPEND	= 130
SYS_SIGALTSTACK	= 131
SYS_UTIME	= 132
SYS_MKNOD	= 133
SYS_USELIB	= 134
SYS_PERSONALITY	= 135
SYS_USTAT	= 136
SYS_STATFS	= 137
SYS_FSTATFS	= 138
SYS_SYSFS	= 139
SYS_GETPRIORITY	= 140
SYS_SETPRIORITY	= 141
SYS_SCHED_SETPARAM	= 142
SYS_SCHED_GETPARAM	= 143
SYS_SCHED_SETSCHEDULER	= 144
SYS_SCHED_GETSCHEDULER	= 145
SYS_SCHED_GET_PRIORITY_MAX	= 146
SYS_SCHED_GET_PRIORITY_MIN	= 147
SYS_SCHED_RR_GET_INTERVAL	= 148
SYS_MLOCK	= 149
SYS_MUNLOCK	= 150
SYS_MLOCKALL	= 151
SYS_MUNLOCKALL	= 152
SYS_VHANGUP	= 153
SYS_MODIFY_LDT	= 154
SYS_PIVOT_ROOT	= 155
SYS__SYSCTL	= 156
SYS_PRCTL	= 157
SYS_ARCH_PRCTL	= 158
SYS_ADJTIMEX	= 159
SYS_SETRLIMIT	= 160
SYS_CHROOT	= 161
SYS_SYNC	= 162
SYS_ACCT	= 163
SYS_SETTIMEOFDAY	= 164
SYS_MOUNT	= 165
SYS_UMOUNT2	= 166
SYS_SWAPON	= 167
SYS_SWAPOFF	= 168
SYS_REBOOT	= 169
SYS_SETHOSTNAME	= 170
SYS_SETDOMAINNAME	= 171
SYS_IOPL	= 172
SYS_IOPERM	= 173
SYS_CREATE_MODULE	= 174
SYS_INIT_MODULE	= 175
SYS_DELETE_MODULE	= 176
SYS_GET_KERNEL_SYMS	= 177
SYS_QUERY_MODULE	= 178

SYS_QUOTACTL	= 179
SYS_NFSSERVCTL	= 180
SYS_GETPMSG	= 181
SYS_PUTPMSG	= 182
SYS_AFS_SYSCALL	= 183
SYS_TUXCALL	= 184
SYS_SECURITY	= 185
SYS_GETTID	= 186
SYS_READAHEAD	= 187
SYS_SETXATTR	= 188
SYS_LSETXATTR	= 189
SYS_FSETXATTR	= 190
SYS_GETXATTR	= 191
SYS_LGETXATTR	= 192
SYS_FGETXATTR	= 193
SYS_LISTXATTR	= 194
SYS_LLISTXATTR	= 195
SYS_FLISTXATTR	= 196
SYS_REMOVEXATTR	= 197
SYS_LREMOVEXATTR	= 198
SYS_FREMOVEXATTR	= 199
SYS_TKILL	= 200
SYS_TIME	= 201
SYS_FUTEX	= 202
SYS_SCHED_SETAFFINITY	= 203
SYS_SCHED_GETAFFINITY	= 204
SYS_SET_THREAD_AREA	= 205
SYS_IO_SETUP	= 206
SYS_IO_DESTROY	= 207
SYS_IO_GETEVENTS	= 208
SYS_IO_SUBMIT	= 209
SYS_IO_CANCEL	= 210
SYS_GET_THREAD_AREA	= 211
SYS_LOOKUP_DCOOKIE	= 212
SYS_EPOLL_CREATE	= 213
SYS_EPOLL_CTL_OLD	= 214
SYS_EPOLL_WAIT_OLD	= 215
SYS_REMAP_FILE_PAGES	= 216
SYS_GETDENTS64	= 217
SYS_SET_TID_ADDRESS	= 218
SYS_RESTART_SYSCALL	= 219
SYS_SEMTIMEDOP	= 220
SYS_FADVISE64	= 221
SYS_TIMER_CREATE	= 222
SYS_TIMER_SETTIME	= 223
SYS_TIMER_GETTIME	= 224
SYS_TIMER_GETOVERRUN	= 225
SYS_TIMER_DELETE	= 226
SYS_CLOCK_SETTIME	= 227
SYS_CLOCK_GETTIME	= 228

SYS_CLOCK_GETRES	= 229
SYS_CLOCK_NANOSLEEP	= 230
SYS_EXIT_GROUP	= 231
SYS_EPOLL_WAIT	= 232
SYS_EPOLL_CTL	= 233
SYS_TGKILL	= 234
SYS_UTIMES	= 235
SYS_VSERVER	= 236
SYS_MBIND	= 237
SYS_SET_MEMPOLICY	= 238
SYS_GET_MEMPOLICY	= 239
SYS_MQ_OPEN	= 240
SYS_MQ_UNLINK	= 241
SYS_MQ_TIMEDSEND	= 242
SYS_MQ_TIMEDRECEIVE	= 243
SYS_MQ_NOTIFY	= 244
SYS_MQ_GETSETATTR	= 245
SYS_KEXEC_LOAD	= 246
SYS_WAITID	= 247
SYS_ADD_KEY	= 248
SYS_REQUEST_KEY	= 249
SYS_KEYCTL	= 250
SYS_IOPRIO_SET	= 251
SYS_IOPRIO_GET	= 252
SYS_INOTIFY_INIT	= 253
SYS_INOTIFY_ADD_WATCH	= 254
SYS_INOTIFY_RM_WATCH	= 255
SYS_MIGRATE_PAGES	= 256
SYS_OPENAT	= 257
SYS_MKDIRAT	= 258
SYS_MKNODAT	= 259
SYS_FCHOWNAT	= 260
SYS_FUTIMESAT	= 261
SYS_NEWFSTATAT	= 262
SYS_UNLINKAT	= 263
SYS_RENAMEAT	= 264
SYS_LINKAT	= 265
SYS_SYMLINKAT	= 266
SYS_READLINKAT	= 267
SYS_FCHMODAT	= 268
SYS_FACCESSAT	= 269
SYS_PSELECT6	= 270
SYS_PPOLL	= 271
SYS_UNSHARE	= 272
SYS_SET_ROBUST_LIST	= 273
SYS_GET_ROBUST_LIST	= 274
SYS_SPLICE	= 275
SYS_TEE	= 276
SYS_SYNC_FILE_RANGE	= 277
SYS_VMSPLICE	= 278

```
SYS_MOVE_PAGES          = 279
SYS_UTIMENSAT           = 280
SYS_EPOLL_PWAIT         = 281
SYS_SIGNALFD            = 282
SYS_TIMERFD_CREATE     = 283
SYS_EVENTFD             = 284
SYS_FALLOCATE           = 285
SYS_TIMERFD_SETTIME    = 286
SYS_TIMERFD_GETTIME    = 287
SYS_ACCEPT4             = 288
SYS_SIGNALFD4           = 289
SYS_EVENTFD2            = 290
SYS_EPOLL_CREATE1      = 291
SYS_DUP3                 = 292
SYS_PIPE2                = 293
SYS_INOTIFY_INIT1      = 294
SYS_PREADV               = 295
SYS_PWRITEV             = 296
SYS_RT_TGSIGQUEUEINFO  = 297
SYS_PERF_EVENT_OPEN    = 298
SYS_RECVMSG             = 299
SYS_FANOTIFY_INIT      = 300
SYS_FANOTIFY_MARK      = 301
SYS_PRLIMIT64          = 302
```

```
)
```

```
const (
    SizeofSockaddrInet4   = 0x10
    SizeofSockaddrInet6   = 0x1c
    SizeofSockaddrAny     = 0x70
    SizeofSockaddrUnix    = 0x6e
    SizeofSockaddrLinklayer = 0x14
    SizeofSockaddrNetlink = 0xc
    SizeofLinger           = 0x8
    SizeofIPMreq           = 0x8
    SizeofIPMreqn          = 0xc
    SizeofIPv6Mreq         = 0x14
    SizeofMsgHdr           = 0x38
    SizeofCmsgHdr          = 0x10
    SizeofInet4Pktinfo     = 0xc
    SizeofInet6Pktinfo     = 0x14
    SizeofUcred            = 0xc
```

```
)
```

```
const (
    IFA_UNSPEC             = 0x0
    IFA_ADDRESS            = 0x1
    IFA_LOCAL              = 0x2
    IFA_LABEL              = 0x3
    IFA_BROADCAST         = 0x4
```

```
IFA_ANYCAST           = 0x5
IFA_CACHEINFO         = 0x6
IFA_MULTICAST         = 0x7
IFLA_UNSPEC           = 0x0
IFLA_ADDRESS          = 0x1
IFLA_BROADCAST        = 0x2
IFLA_IFNAME           = 0x3
IFLA_MTU              = 0x4
IFLA_LINK             = 0x5
IFLA_QDISC            = 0x6
IFLA_STATS            = 0x7
IFLA_COST             = 0x8
IFLA_PRIORITY         = 0x9
IFLA_MASTER           = 0xa
IFLA_WIRELESS         = 0xb
IFLA_PROTINFO         = 0xc
IFLA_TXQLEN          = 0xd
IFLA_MAP              = 0xe
IFLA_WEIGHT           = 0xf
IFLA_OPERSTATE        = 0x10
IFLA_LINKMODE         = 0x11
IFLA_LINKINFO        = 0x12
IFLA_NET_NS_PID      = 0x13
IFLA_IFALIAS         = 0x14
IFLA_MAX              = 0x1c
RT_SCOPE_UNIVERSE    = 0x0
RT_SCOPE_SITE        = 0xc8
RT_SCOPE_LINK        = 0xfd
RT_SCOPE_HOST        = 0xfe
RT_SCOPE_NOWHERE     = 0xff
RT_TABLE_UNSPEC      = 0x0
RT_TABLE_COMPAT      = 0xfc
RT_TABLE_DEFAULT     = 0xfd
RT_TABLE_MAIN        = 0xfe
RT_TABLE_LOCAL       = 0xff
RT_TABLE_MAX         = 0xffffffff
RTA_UNSPEC           = 0x0
RTA_DST              = 0x1
RTA_SRC              = 0x2
RTA_IIF              = 0x3
RTA_OIF              = 0x4
RTA_GATEWAY          = 0x5
RTA_PRIORITY         = 0x6
RTA_PREFSRC          = 0x7
RTA_METRICS          = 0x8
RTA_MULTIPATH        = 0x9
RTA_FLOW             = 0xb
RTA_CACHEINFO        = 0xc
RTA_TABLE            = 0xf
RTN_UNSPEC           = 0x0
```

```
RTN_UNICAST      = 0x1
RTN_LOCAL        = 0x2
RTN_BROADCAST    = 0x3
RTN_ANYCAST      = 0x4
RTN_MULTICAST    = 0x5
RTN_BLACKHOLE    = 0x6
RTN_UNREACHABLE  = 0x7
RTN_PROHIBIT     = 0x8
RTN_THROW        = 0x9
RTN_NAT          = 0xa
RTN_XRESOLVE     = 0xb
SizeofNlMsgHdr   = 0x10
SizeofNlMsgerr   = 0x14
SizeofRtGenmsg   = 0x1
SizeofNlAttr     = 0x4
SizeofRtAttr     = 0x4
SizeofIfInfomsg  = 0x10
SizeofIfAddrmsg  = 0x8
SizeofRtMsg      = 0xc
SizeofRtNextHop  = 0x8
)
```

```
const (
    SizeofSockFilter = 0x8
    SizeofSockFprog  = 0x10
)
```

```
const (
    VINTR      = 0x0
    VQUIT      = 0x1
    VERASE     = 0x2
    VKILL      = 0x3
    VEOF       = 0x4
    VTIME      = 0x5
    VMIN       = 0x6
    VSWTC      = 0x7
    VSTART     = 0x8
    VSTOP      = 0x9
    VSUSP     = 0xa
    VEOL       = 0xb
    VREPRINT   = 0xc
    VDISCARD   = 0xd
    VWERASE    = 0xe
    VLNEXT     = 0xf
    VEOL2      = 0x10
    IGNBRK     = 0x1
    BRKINT     = 0x2
    IGNPAR     = 0x4
    PARMRK     = 0x8
    INPCK      = 0x10
)
```

ISTRIP	= 0x20
INLCR	= 0x40
IGNCR	= 0x80
ICRNL	= 0x100
IUCLC	= 0x200
IXON	= 0x400
IXANY	= 0x800
IXOFF	= 0x1000
IMAXBEL	= 0x2000
IUTF8	= 0x4000
OPOST	= 0x1
OLCUC	= 0x2
ONLCR	= 0x4
OCRNL	= 0x8
ONOCR	= 0x10
ONLRET	= 0x20
OFILL	= 0x40
OFDEL	= 0x80
B0	= 0x0
B50	= 0x1
B75	= 0x2
B110	= 0x3
B134	= 0x4
B150	= 0x5
B200	= 0x6
B300	= 0x7
B600	= 0x8
B1200	= 0x9
B1800	= 0xa
B2400	= 0xb
B4800	= 0xc
B9600	= 0xd
B19200	= 0xe
B38400	= 0xf
CSIZE	= 0x30
CS5	= 0x0
CS6	= 0x10
CS7	= 0x20
CS8	= 0x30
CSTOPB	= 0x40
CREAD	= 0x80
PARENB	= 0x100
PARODD	= 0x200
HUPCL	= 0x400
CLOCAL	= 0x800
B57600	= 0x1001
B115200	= 0x1002
B230400	= 0x1003
B460800	= 0x1004
B500000	= 0x1005

```
B576000 = 0x1006
B921600 = 0x1007
B1000000 = 0x1008
B1152000 = 0x1009
B1500000 = 0x100a
B2000000 = 0x100b
B2500000 = 0x100c
B3000000 = 0x100d
B3500000 = 0x100e
B4000000 = 0x100f
ISIG      = 0x1
ICANON    = 0x2
XCASE     = 0x4
ECHO      = 0x8
ECHOE     = 0x10
ECHOK     = 0x20
ECHONL    = 0x40
NOFLSH    = 0x80
TOSTOP    = 0x100
ECHOCTL   = 0x200
ECHOPRT   = 0x400
ECHOKE    = 0x800
FLUSHO    = 0x1000
PENDIN    = 0x4000
IEXTEN    = 0x8000
TCGETS    = 0x5401
TCSETS    = 0x5402
```

```
)
```

```
const ImplementsGetwd = true
```

```
const (  
    PathMax = 0x1000  
)
```

```
const SizeofInotifyEvent = 0x10
```

Variables

```
var (  
    Stdin  = 0  
    Stdout = 1  
    Stderr = 2  
)  
  
var ForkLock sync.RWMutex  
  
var SocketDisableIPv6 bool
```

For testing: clients can set this flag to force creation of IPv6 sockets to return EAFNOSUPPORT.

func [Accept](#)

```
func Accept(fd int) (nfd int, sa Sockaddr, err error)
```

func [Access](#)

```
func Access(path string, mode uint32) (err error)
```

func [Acct](#)

```
func Acct(path string) (err error)
```

func [Adjtimex](#)

```
func Adjtimex(buf *Timex) (state int, err error)
```

func [AttachLsf](#)

```
func AttachLsf(fd int, i []SockFilter) error
```

func [Bind](#)

```
func Bind(fd int, sa Sockaddr) (err error)
```

func [BindToDevice](#)

```
func BindToDevice(fd int, device string) (err error)
```

BindToDevice binds the socket associated with fd to device.

func [Chdir](#)

```
func Chdir(path string) (err error)
```

func [Chmod](#)

```
func Chmod(path string, mode uint32) (err error)
```

func [Chown](#)

```
func Chown(path string, uid int, gid int) (err error)
```

func [Chroot](#)

```
func Chroot(path string) (err error)
```

func [Clearenv](#)

func Clearenv()

func Close

```
func Close(fd int) (err error)
```

func CloseOnExec

```
func CloseOnExec(fd int)
```

func CmsgLen

```
func CmsgLen(dataLen int) int
```

CmsgLen returns the value to store in the Len field of the CmsgHdr structure, taking into account any necessary alignment.

func CmsgSpace

```
func CmsgSpace(dataLen int) int
```

CmsgSpace returns the number of bytes an ancillary element with payload of the passed data length occupies.

func [Connect](#)

```
func Connect(fd int, sa Sockaddr) (err error)
```

func [Creat](#)

```
func Creat(path string, mode uint32) (fd int, err error)
```

func DetachLsf

func DetachLsf(fd int) error

func Dup

```
func Dup(oldfd int) (fd int, err error)
```

func Dup2

```
func Dup2(oldfd int, newfd int) (err error)
```

func Environ

```
func Environ() []string
```

func EpollCreate

```
func EpollCreate(size int) (fd int, err error)
```

func [EpollCreate1](#)

```
func EpollCreate1(flag int) (fd int, err error)
```

func [EpollCtl](#)

```
func EpollCtl(epfd int, op int, fd int, event *EpollEvent) (err error)
```

func [EpollWait](#)

```
func EpollWait(epfd int, events []EpollEvent, msec int) (n int, err
```

func [Exec](#)

```
func Exec(argv0 string, argv []string, envv []string) (err error)
```

Ordinary exec.

func [Exit](#)

```
func Exit(code int)
```

func [Faccessat](#)

func Faccessat(dirfd int, path string, mode uint32, flags int) (err

func Fallocate

func Fallocate(fd int, mode uint32, off int64, len int64) (err error)

func [Fchdir](#)

```
func Fchdir(fd int) (err error)
```

func [Fchmod](#)

func Fchmod(fd int, mode uint32) (err error)

func [Fchmodat](#)

`func Fchmodat(dirfd int, path string, mode uint32, flags int) (err e`

func [Fchown](#)

```
func Fchown(fd int, uid int, gid int) (err error)
```

func [Fchownat](#)

```
func Fchownat(dirfd int, path string, uid int, gid int, flags int) (
```

func [Fdatasync](#)

```
func Fdatasync(fd int) (err error)
```

func Flock

```
func Flock(fd int, how int) (err error)
```

func [ForkExec](#)

```
func ForkExec(argv0 string, argv []string, attr *ProcAttr) (pid int,
```

Combination of fork and exec, careful to be thread safe.

func Fstat

```
func Fstat(fd int, stat *Stat_t) (err error)
```

func [Fstatfs](#)

func Fstatfs(fd int, buf *Statfs_t) (err error)

func [Fsync](#)

```
func Fsync(fd int) (err error)
```

func [Ftruncate](#)

```
func Ftruncate(fd int, length int64) (err error)
```

func Futimes

```
func Futimes(fd int, tv []Timeval) (err error)
```

func [Futimesat](#)

```
func Futimesat(dirfd int, path string, tv []Timeval) (err error)
```

```
sys futimesat(dirfd int, path *byte, times *[2]Timeval) (err error)
```

func [Getcwd](#)

```
func Getcwd(buf []byte) (n int, err error)
```

func [Getdents](#)

```
func Getdents(fd int, buf []byte) (n int, err error)
```

func [Getegid](#)

```
func Getegid() (egid int)
```

func [Getenv](#)

```
func Getenv(key string) (value string, found bool)
```

func Geteuid

```
func Geteuid() (euid int)
```

func [Getgid](#)

```
func Getgid() (gid int)
```

func Getgroups

```
func Getgroups() (gids []int, err error)
```

func [Getpagesize](#)

```
func Getpagesize() int
```

func [Getpgid](#)

```
func Getpgid(pid int) (pgid int, err error)
```

func [Getpgrp](#)

```
func Getpgrp() (pid int)
```

func [Getpid](#)

```
func Getpid() (pid int)
```

func [Getppid](#)

```
func Getppid() (ppid int)
```

func [Getrlimit](#)

```
func Getrlimit(resource int, rlim *Rlimit) (err error)
```

func Getrusage

```
func Getrusage(who int, rusage *Rusage) (err error)
```

func [GetsockoptInet4Addr](#)

func GetsockoptInet4Addr(fd, level, opt int) (value [4]byte, err error)

func [GetsockoptInt](#)

```
func GetsockoptInt(fd, level, opt int) (value int, err error)
```

func [Gettid](#)

```
func Gettid() (tid int)
```

func [Gettimeofday](#)

```
func Gettimeofday(tv *Timeval) (err error)
```

func Getuid

```
func Getuid() (uid int)
```

func [Getwd](#)

func Getwd() (wd string, err error)

sys Getcwd(buf []byte) (n int, err error)

func InotifyAddWatch

```
func InotifyAddWatch(fd int, pathname string, mask uint32) (watchdes
```

func InotifyInit

```
func InotifyInit() (fd int, err error)
```

func [InotifyInit1](#)

```
func InotifyInit1(flags int) (fd int, err error)
```

func InotifyRmWatch

```
func InotifyRmWatch(fd int, watchdesc uint32) (success int, err error)
```

func Ioperm

```
func Ioperm(from int, num int, on int) (err error)
```

func Iopl

```
func Iopl(level int) (err error)
```

func [Kill](#)

```
func Kill(pid int, sig Signal) (err error)
```

func [Klogctl](#)

```
func Klogctl(typ int, buf []byte) (n int, err error)
```

func [Lchown](#)

```
func Lchown(path string, uid int, gid int) (err error)
```

func [Link](#)

```
func Link(oldpath string, newpath string) (err error)
```

func Listen

```
func Listen(s int, n int) (err error)
```

func [LsfSocket](#)

```
func LsfSocket(ifindex, proto int) (int, error)
```

func Lstat

```
func Lstat(path string, stat *Stat_t) (err error)
```

func [Madvise](#)

```
func Madvise(b []byte, advice int) (err error)
```

func [Mkdir](#)

```
func Mkdir(path string, mode uint32) (err error)
```

func [Mkdirat](#)

```
func Mkdirat(dirfd int, path string, mode uint32) (err error)
```

func [Mkfifo](#)

```
func Mkfifo(path string, mode uint32) (err error)
```

func [Mknod](#)

```
func Mknod(path string, mode uint32, dev int) (err error)
```

func [Mknodat](#)

```
func Mknodat(dirfd int, path string, mode uint32, dev int) (err error)
```

func Mlock

```
func Mlock(b []byte) (err error)
```

func Mlockall

func Mlockall(flags int) (err error)

func [Mmap](#)

func Mmap(fd int, offset int64, length int, prot int, flags int) (da

func [Mount](#)

```
func Mount(source string, target string, fstype string, flags uintpt
```

```
sys mount(source string, target string, fstype string, flags uintptr, data *byte) (err  
error)
```

func [Mprotect](#)

```
func Mprotect(b []byte, prot int) (err error)
```

func [Munlock](#)

```
func Munlock(b []byte) (err error)
```

func [Munlockall](#)

func Munlockall() (err error)

func [Munmap](#)

func Munmap(b []byte) (err error)

func [Nanosleep](#)

```
func Nanosleep(time *Timespec, leftover *Timespec) (err error)
```

func [NetlinkRIB](#)

```
func NetlinkRIB(proto, family int) ([]byte, error)
```

NetlinkRIB returns routing information base, as known as RIB, which consists of network facility information, states and parameters.

func [Open](#)

```
func Open(path string, mode int, perm uint32) (fd int, err error)
```

```
sys open(path string, mode int, perm uint32) (fd int, err error)
```

func [Openat](#)

```
func Openat(dirfd int, path string, flags int, mode uint32) (fd int,  
sys openat(dirfd int, path string, flags int, mode uint32) (fd int, err error)
```

func ParseDirent

```
func ParseDirent(buf []byte, max int, names []string) (consumed int,
```

func ParseNetlinkMessage

```
func ParseNetlinkMessage(buf []byte) ([]NetlinkMessage, error)
```

ParseNetlinkMessage parses buf as netlink messages and returns the slice containing the NetlinkMessage structs.

func ParseNetlinkRouteAttr

```
func ParseNetlinkRouteAttr(msg *NetlinkMessage) ([]NetlinkRouteAttr,
```

ParseNetlinkRouteAttr parses msg's payload as netlink route attributes and returns the slice containing the NetlinkRouteAttr structs.

func ParseSocketControlMessage

```
func ParseSocketControlMessage(buf []byte) ([]SocketControlMessage,
```

func [ParseUnixRights](#)

```
func ParseUnixRights(msg *SocketControlMessage) ([]int, error)
```

ParseUnixRights decodes a socket control message that contains an integer array of open file descriptors from another process.

func Pause

func Pause() (err error)

func [Pipe](#)

func Pipe(p []int) (err error)

sysnb pipe(p *[2]_C_int) (err error)

func PivotRoot

```
func PivotRoot(newroot string, putold string) (err error)
```

func Pread

```
func Pread(fd int, p []byte, offset int64) (n int, err error)
```

func PtraceAttach

func PtraceAttach(pid int) (err error)

func [PtraceCont](#)

```
func PtraceCont(pid int, signal int) (err error)
```

func [PtraceDetach](#)

```
func PtraceDetach(pid int) (err error)
```

func [PtraceGetEventMsg](#)

```
func PtraceGetEventMsg(pid int) (msg uint, err error)
```

func [PtraceGetRegs](#)

```
func PtraceGetRegs(pid int, regsout *PtraceRegs) (err error)
```

func PtracePeekData

```
func PtracePeekData(pid int, addr uintptr, out []byte) (count int, e
```

func PtracePeekText

```
func PtracePeekText(pid int, addr uintptr, out []byte) (count int, e
```

func PtracePokeData

```
func PtracePokeData(pid int, addr uintptr, data []byte) (count int,
```

func PtracePokeText

```
func PtracePokeText(pid int, addr uintptr, data []byte) (count int,
```

func PtraceSetOptions

```
func PtraceSetOptions(pid int, options int) (err error)
```

func [PtraceSetRegs](#)

```
func PtraceSetRegs(pid int, regs *PtraceRegs) (err error)
```

func [PtraceSingleStep](#)

```
func PtraceSingleStep(pid int) (err error)
```

func [Pwrite](#)

```
func Pwrite(fd int, p []byte, offset int64) (n int, err error)
```

func [RawSyscall](#)

```
func RawSyscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err Errno
```

func [RawSyscall6](#)

```
func RawSyscall6(trap, a1, a2, a3, a4, a5, a6 uintptr) (r1, r2 uintptr)
```

func Read

```
func Read(fd int, p []byte) (n int, err error)
```

func [ReadDirent](#)

```
func ReadDirent(fd int, buf []byte) (n int, err error)
```

func [Readlink](#)

```
func Readlink(path string, buf []byte) (n int, err error)
```

func [Reboot](#)

```
func Reboot(cmd int) (err error)
```

```
sys reboot(magic1 uint, magic2 uint, cmd int, arg string) (err error)
```

func Recvfrom

func Recvfrom(fd int, p []byte, flags int) (n int, from Sockaddr, err error)

func Recvmsg

func Recvmsg(fd int, p, oob []byte, flags int) (n, oobn int, recvfla

func [Rename](#)

```
func Rename(oldpath string, newpath string) (err error)
```

func [Renameat](#)

```
func Renameat(olddirfd int, oldpath string, newdirfd int, newpath st
```

func [Rmdir](#)

```
func Rmdir(path string) (err error)
```

func [Seek](#)

```
func Seek(fd int, offset int64, whence int) (off int64, err error)
```

func [Select](#)

```
func Select(nfd int, r *FdSet, w *FdSet, e *FdSet, timeout *Timeval)
```

func [Sendfile](#)

```
func Sendfile(outfd int, infd int, offset *int64, count int) (writte
```

func [Sendmsg](#)

func Sendmsg(fd int, p, oob []byte, to Sockaddr, flags int) (err err

func [Sendto](#)

```
func Sendto(fd int, p []byte, flags int, to Sockaddr) (err error)
```

func [SetLsfPromisc](#)

```
func SetLsfPromisc(name string, m bool) error
```

func [SetNonblock](#)

```
func SetNonblock(fd int, nonblocking bool) (err error)
```

func [Setdomainname](#)

func Setdomainname(p []byte) (err error)

func [Setenv](#)

func Setenv(key, value string) error

func [Setfsgid](#)

```
func Setfsgid(gid int) (err error)
```

func [Setfsuid](#)

```
func Setfsuid(uid int) (err error)
```

func [Setgid](#)

```
func Setgid(gid int) (err error)
```

func Setgroups

```
func Setgroups(gids []int) (err error)
```

func [Sethostname](#)

```
func Sethostname(p []byte) (err error)
```

func [Setpgid](#)

func Setpgid(pid int, pgid int) (err error)

func [Setregid](#)

```
func Setregid(rgid int, egid int) (err error)
```

func [Setresgid](#)

func Setresgid(rgid int, egid int, sgid int) (err error)

func [Setresuid](#)

```
func Setresuid(ruid int, euid int, suid int) (err error)
```

func [Setreuid](#)

```
func Setreuid(ruid int, euid int) (err error)
```

func [Setrlimit](#)

```
func Setrlimit(resource int, rlim *Rlimit) (err error)
```

func [Setsid](#)

```
func Setsid() (pid int, err error)
```

func [SetsockoptIPMreq](#)

```
func SetsockoptIPMreq(fd, level, opt int, mreq *IPMreq) (err error)
```

func [SetsockoptIPMreqn](#)

```
func SetsockoptIPMreqn(fd, level, opt int, mreq *IPMreqn) (err error
```

func [SetsockoptIPv6Mreq](#)

```
func SetsockoptIPv6Mreq(fd, level, opt int, mreq *IPv6Mreq) (err err
```

func [SetsockoptInet4Addr](#)

func SetsockoptInet4Addr(fd, level, opt int, value [4]byte) (err err

func [SetsockoptInt](#)

```
func SetsockoptInt(fd, level, opt int, value int) (err error)
```

func [SetsockoptLinger](#)

```
func SetsockoptLinger(fd, level, opt int, l *Linger) (err error)
```

func [SetsockoptString](#)

```
func SetsockoptString(fd, level, opt int, s string) (err error)
```

func [SetsockoptTimeval](#)

```
func SetsockoptTimeval(fd, level, opt int, tv *Timeval) (err error)
```

func [Settimeofday](#)

```
func Settimeofday(tv *Timeval) (err error)
```

func [Setuid](#)

```
func Setuid(uid int) (err error)
```

func Shutdown

```
func Shutdown(fd int, how int) (err error)
```

func [Socket](#)

```
func Socket(domain, typ, proto int) (fd int, err error)
```

func [Socketpair](#)

```
func Socketpair(domain, typ, proto int) (fd [2]int, err error)
```

func [Splice](#)

```
func Splice(rfd int, roff *int64, wfd int, woff *int64, len int, fla
```

func [StartProcess](#)

```
func StartProcess(argv0 string, argv []string, attr *ProcAttr) (pid
```

StartProcess wraps ForkExec for package os.

func [Stat](#)

```
func Stat(path string, stat *Stat_t) (err error)
```

func [Statfs](#)

```
func Statfs(path string, buf *Statfs_t) (err error)
```

func [StringBytePtr](#)

```
func StringBytePtr(s string) *byte
```

StringBytePtr returns a pointer to a NUL-terminated array of bytes containing the text of s.

func [StringByteSlice](#)

```
func StringByteSlice(s string) []byte
```

StringByteSlice returns a NUL-terminated slice of bytes containing the text of s.

func [StringSlicePtr](#)

```
func StringSlicePtr(ss []string) []*byte
```

Convert array of string to array of NUL-terminated byte pointer.

func [SymLink](#)

```
func SymLink(oldpath string, newpath string) (err error)
```

func Sync

func Sync()

func SyncFileRange

func SyncFileRange(fd int, off int64, n int64, flags int) (err error

func Syscall

```
func Syscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err Errno)
```

func [Syscall6](#)

```
func Syscall6(trap, a1, a2, a3, a4, a5, a6 uintptr) (r1, r2 uintptr,
```

func [Sysinfo](#)

```
func Sysinfo(info *Sysinfo_t) (err error)
```

func Tee

```
func Tee(rfd int, wfd int, len int, flags int) (n int64, err error)
```

func Tgkill

```
func Tgkill(tgid int, tid int, sig Signal) (err error)
```

func [Times](#)

```
func Times(tms *Tms) (ticks uintptr, err error)
```

func TimespecToNsec

```
func TimespecToNsec(ts Timespec) int64
```

func [TimevalToNsec](#)

```
func TimevalToNsec(tv Timeval) int64
```

func [Truncate](#)

```
func Truncate(path string, length int64) (err error)
```

func Umask

func Umask(mask int) (oldmask int)

func [Uname](#)

```
func Uname(buf *Utsname) (err error)
```

func [UnixCredentials](#)

```
func UnixCredentials(ucred *Ucred) []byte
```

UnixCredentials encodes credentials into a socket control message for sending to another process. This can be used for authentication.

func [UnixRights](#)

```
func UnixRights(fds ...int) []byte
```

UnixRights encodes a set of open file descriptors into a socket control message for sending to another process.

func [Unlink](#)

```
func Unlink(path string) (err error)
```

func [Unlinkat](#)

```
func Unlinkat(dirfd int, path string) (err error)
```

func Unmount

```
func Unmount(target string, flags int) (err error)
```

func [Unshare](#)

```
func Unshare(flags int) (err error)
```

func [Ustat](#)

func Ustat(dev int, ubuf *Ustat_t) (err error)

func [Utime](#)

```
func Utime(path string, buf *Utimbuf) (err error)
```

func [Utimes](#)

```
func Utimes(path string, tv []Timeval) (err error)
```

```
sys utimes(path string, times *[2]Timeval) (err error)
```

func [Wait4](#)

```
func Wait4(pid int, wstatus *WaitStatus, options int, rusage *Rusage
```

```
sys wait4(pid int, wstatus *_C_int, options int, rusage *Rusage) (wpid int, err  
error)
```

func [Write](#)

```
func Write(fd int, p []byte) (n int, err error)
```

type [CmsgHdr](#)

```
type CmsgHdr struct {  
    Len          uint64  
    Level        int32  
    Type         int32  
    X__cmsg_data [0]byte  
}
```

func (*CmsgHdr) [SetLen](#)

```
func (cmsg *CmsgHdr) SetLen(length int)
```

type [Credential](#)

```
type Credential struct {  
    Uid    uint32    // User ID.  
    Gid    uint32    // Group ID.  
    Groups []uint32 // Supplementary group IDs.  
}
```

Credential holds user and group identities to be assumed by a child process started by `StartProcess`.

type [Dirent](#)

```
type Dirent struct {  
    Ino      uint64  
    Off      int64  
    Reclen   uint16  
    Type     uint8  
    Name     [256]int8  
    Pad_cgo_0 [5]byte  
}
```

type EpollEvent

```
type EpollEvent struct {  
    Events uint32  
    Fd      int32  
    Pad     int32  
}
```

type [Errno](#)

```
type Errno uintptr
```

An Errno is an unsigned number describing an error condition. It implements the error interface. The zero Errno is by convention a non-error, so code to convert from Errno to error should use:

```
err = nil
if errno != 0 {
    err = errno
}
```

func (Errno) [Error](#)

```
func (e Errno) Error() string
```

func (Errno) [Temporary](#)

```
func (e Errno) Temporary() bool
```

func (Errno) [Timeout](#)

```
func (e Errno) Timeout() bool
```

type [FdSet](#)

```
type FdSet struct {  
    Bits [16]int64  
}
```

type [Fsid](#)

```
type Fsid struct {  
    X__val [2]int32  
}
```

type [IPMreq](#)

```
type IPMreq struct {  
    Multiaddr [4]byte /* in_addr */  
    Interface [4]byte /* in_addr */  
}
```

func [GetsockoptIPMreq](#)

```
func GetsockoptIPMreq(fd, level, opt int) (*IPMreq, error)
```

type [IPMreqn](#)

```
type IPMreqn struct {  
    Multiaddr [4]byte /* in_addr */  
    Address   [4]byte /* in_addr */  
    Ifindex   int32  
}
```

func [GetsockoptIPMreqn](#)

```
func GetsockoptIPMreqn(fd, level, opt int) (*IPMreqn, error)
```

type [IPv6Mreq](#)

```
type IPv6Mreq struct {  
    Multiaddr [16]byte /* in6_addr */  
    Interface uint32  
}
```

func [GetsockoptIPv6Mreq](#)

```
func GetsockoptIPv6Mreq(fd, level, opt int) (*IPv6Mreq, error)
```

type [IfAddrmsg](#)

```
type IfAddrmsg struct {  
    Family      uint8  
    Prefixlen   uint8  
    Flags       uint8  
    Scope       uint8  
    Index       uint32  
}
```

type IfInfomsg

```
type IfInfomsg struct {  
    Family      uint8  
    X__ifi_pad  uint8  
    Type        uint16  
    Index       int32  
    Flags       uint32  
    Change      uint32  
}
```

type [Inet4Pktinfo](#)

```
type Inet4Pktinfo struct {  
    Ifindex  int32  
    Spec_dst [4]byte /* in_addr */  
    Addr     [4]byte /* in_addr */  
}
```

type Inet6Pktinfo

```
type Inet6Pktinfo struct {  
    Addr    [16]byte /* in6_addr */  
    Ifindex uint32  
}
```

type InotifyEvent

```
type InotifyEvent struct {  
    Wd      int32  
    Mask    uint32  
    Cookie  uint32  
    Len     uint32  
    Name    [0]byte  
}
```

type [Iovec](#)

```
type Iovec struct {  
    Base *byte  
    Len  uint64  
}
```

func (*Iovec) [SetLen](#)

```
func (iov *Iovec) SetLen(length int)
```

type Linger

```
type Linger struct {  
    Onoff int32  
    Linger int32  
}
```

type [Msghdr](#)

```
type Msghdr struct {
    Name      *byte
    Namelen   uint32
    Pad_cgo_0 [4]byte
    Iov       *Iovec
    Iovlen    uint64
    Control   *byte
    Controllen uint64
    Flags     int32
    Pad_cgo_1 [4]byte
}
```

func (*Msghdr) [SetControllen](#)

```
func (msghdr *Msghdr) SetControllen(length int)
```

type [NetlinkMessage](#)

```
type NetlinkMessage struct {  
    Header NlMsgHdr  
    Data []byte  
}
```

NetlinkMessage represents the netlink message.

type [NetlinkRouteAttr](#)

```
type NetlinkRouteAttr struct {  
    Attr RtAttr  
    Value []byte  
}
```

NetlinkRouteAttr represents the netlink route attribute.

type [NetlinkRouteRequest](#)

```
type NetlinkRouteRequest struct {  
    Header NlMsgHdr  
    Data    RtGenmsg  
}
```

NetlinkRouteRequest represents the request message to receive routing and link states from the kernel.

type [NlAttr](#)

```
type NlAttr struct {  
    Len  uint16  
    Type uint16  
}
```

type [NlMsgerr](#)

```
type NlMsgerr struct {  
    Error int32  
    Msg   NlMsghdr  
}
```

type [NlMsgHdr](#)

```
type NlMsgHdr struct {  
    Len    uint32  
    Type   uint16  
    Flags  uint16  
    Seq    uint32  
    Pid    uint32  
}
```

type [ProcAttr](#)

```
type ProcAttr struct {
    Dir    string    // Current working directory.
    Env    []string   // Environment.
    Files  []uintptr  // File descriptors.
    Sys    *SysProcAttr
}
```

ProcAttr holds attributes that will be applied to a new process started by StartProcess.

type [PtraceRegs](#)

```
type PtraceRegs struct {  
    R15      uint64  
    R14      uint64  
    R13      uint64  
    R12      uint64  
    Rbp      uint64  
    Rbx      uint64  
    R11      uint64  
    R10      uint64  
    R9       uint64  
    R8       uint64  
    Rax      uint64  
    Rcx      uint64  
    Rdx      uint64  
    Rsi      uint64  
    Rdi      uint64  
    Orig_rax uint64  
    Rip      uint64  
    Cs       uint64  
    Eflags   uint64  
    Rsp      uint64  
    Ss       uint64  
    Fs_base  uint64  
    Gs_base  uint64  
    Ds       uint64  
    Es       uint64  
    Fs       uint64  
    Gs       uint64  
}
```

func (*PtraceRegs) [PC](#)

```
func (r *PtraceRegs) PC() uint64
```

func (*PtraceRegs) [SetPC](#)

```
func (r *PtraceRegs) SetPC(pc uint64)
```

type [RawSockaddr](#)

```
type RawSockaddr struct {  
    Family uint16  
    Data   [14]int8  
}
```

type [RawSockaddrAny](#)

```
type RawSockaddrAny struct {  
    Addr RawSockaddr  
    Pad  [96]int8  
}
```

type [RawSockaddrInet4](#)

```
type RawSockaddrInet4 struct {
    Family uint16
    Port    uint16
    Addr    [4]byte /* in_addr */
    Zero    [8]uint8
}
```

type [RawSockaddrInet6](#)

```
type RawSockaddrInet6 struct {
    Family    uint16
    Port      uint16
    Flowinfo  uint32
    Addr      [16]byte /* in6_addr */
    Scope_id  uint32
}
```

type [RawSockaddrLinklayer](#)

```
type RawSockaddrLinklayer struct {  
    Family    uint16  
    Protocol  uint16  
    Ifindex   int32  
    Hatype    uint16  
    Pkttype   uint8  
    Halen     uint8  
    Addr      [8]uint8  
}
```

type [RawSockaddrNetlink](#)

```
type RawSockaddrNetlink struct {  
    Family uint16  
    Pad    uint16  
    Pid    uint32  
    Groups uint32  
}
```

type [RawSockaddrUnix](#)

```
type RawSockaddrUnix struct {  
    Family uint16  
    Path   [108]int8  
}
```

type Rlimit

```
type Rlimit struct {  
    Cur uint64  
    Max uint64  
}
```

type [RtAttr](#)

```
type RtAttr struct {  
    Len  uint16  
    Type uint16  
}
```

type [RtGenmsg](#)

```
type RtGenmsg struct {  
    Family uint8  
}
```

type RtMsg

```
type RtMsg struct {  
    Family    uint8  
    Dst_len   uint8  
    Src_len   uint8  
    Tos       uint8  
    Table     uint8  
    Protocol  uint8  
    Scope     uint8  
    Type      uint8  
    Flags     uint32  
}
```

type [RtNextHop](#)

```
type RtNextHop struct {  
    Len      uint16  
    Flags    uint8  
    Hops     uint8  
    Ifindex  int32  
}
```

type Rusage

```
type Rusage struct {
    Utime    Timeval
    Stime    Timeval
    Maxrss   int64
    Ixrss    int64
    Idrss    int64
    Isrss    int64
    Minflt   int64
    Majflt   int64
    Nswap    int64
    Inblock  int64
    Oublock  int64
    Msgsnd   int64
    Msgrcv   int64
    Nsignals int64
    Nvcsw    int64
    Nivcsw   int64
}
```

type [Signal](#)

```
type Signal int
```

A Signal is a number describing a process signal. It implements the `os.Signal` interface.

func (Signal) [Signal](#)

```
func (s Signal) Signal()
```

func (Signal) [String](#)

```
func (s Signal) String() string
```

type [SockFilter](#)

```
type SockFilter struct {  
    Code uint16  
    Jt    uint8  
    Jf    uint8  
    K     uint32  
}
```

func [LsfJump](#)

```
func LsfJump(code, k, jt, jf int) *SockFilter
```

func [LsfStmt](#)

```
func LsfStmt(code, k int) *SockFilter
```

type [SockFprog](#)

```
type SockFprog struct {  
    Len      uint16  
    Pad_cgo_0 [6]byte  
    Filter    *SockFilter  
}
```

type [Sockaddr](#)

```
type Sockaddr interface {  
    // contains filtered or unexported methods  
}
```

func [Getpeername](#)

```
func Getpeername(fd int) (sa Sockaddr, err error)
```

func [Getsockname](#)

```
func Getsockname(fd int) (sa Sockaddr, err error)
```

type [SockaddrInet4](#)

```
type SockaddrInet4 struct {  
    Port int  
    Addr [4]byte  
    // contains filtered or unexported fields  
}
```

type [SockaddrInet6](#)

```
type SockaddrInet6 struct {
    Port    int
    ZoneId  uint32
    Addr    [16]byte
    // contains filtered or unexported fields
}
```

type [SockaddrLinklayer](#)

```
type SockaddrLinklayer struct {
    Protocol uint16
    Ifindex  int
    Hatype   uint16
    Pkttype  uint8
    Halen    uint8
    Addr     [8]byte
    // contains filtered or unexported fields
}
```

type [SockaddrNetlink](#)

```
type SockaddrNetlink struct {  
    Family uint16  
    Pad    uint16  
    Pid    uint32  
    Groups uint32  
    // contains filtered or unexported fields  
}
```

type [SockaddrUnix](#)

```
type SockaddrUnix struct {  
    Name string  
    // contains filtered or unexported fields  
}
```

type [SocketControlMessage](#)

```
type SocketControlMessage struct {  
    Header CmsgHdr  
    Data   []byte  
}
```

type [Stat_t](#)

```
type Stat_t struct {
    Dev      uint64
    Ino      uint64
    Nlink    uint64
    Mode     uint32
    Uid      uint32
    Gid      uint32
    X__pad0  int32
    Rdev     uint64
    Size     int64
    Blksize  int64
    Blocks   int64
    Atim     Timespec
    Mtim     Timespec
    Ctim     Timespec
    X__unused [3]int64
}
```

type [Statfs_t](#)

```
type Statfs_t struct {
    Type      int64
    Bsize     int64
    Blocks    uint64
    Bfree     uint64
    Bavail    uint64
    Files     uint64
    Ffree     uint64
    Fsid      Fsid
    Namelen   int64
    Fsize     int64
    Flags     int64
    Spare     [4]int64
}
```

type [SysProcAttr](#)

```
type SysProcAttr struct {
    Chroot      string    // Chroot.
    Credential  *Credential // Credential.
   Ptrace      bool      // Enable tracing.
   Setsid      bool      // Create session.
   Setpgid     bool      // Set process group ID to new pid (SYSV
   Setctty     bool      // Set controlling terminal to fd 0
   Noctty      bool      // Detach fd 0 from controlling terminal
   Pdeathsig   Signal    // Signal that the process will get when
}
```

type Sysinfo_t

```
type Sysinfo_t struct {
    Uptime      int64
    Loads       [3]uint64
    Totalram    uint64
    Freeram     uint64
    Sharedram   uint64
    Bufferram   uint64
    Totalswap   uint64
    Freeswap    uint64
    Procs       uint16
    Pad         uint16
    Pad_cgo_0   [4]byte
    Totalhigh   uint64
    Freehigh    uint64
    Unit        uint32
    X_f         [0]byte
    Pad_cgo_1   [4]byte
}
```

type [Termios](#)

```
type Termios struct {
    Iflag      uint32
    Oflag      uint32
    Cflag      uint32
    Lflag      uint32
    Line       uint8
    Cc         [32]uint8
    Pad_cgo_0  [3]byte
    Ispeed     uint32
    Ospeed     uint32
}
```

type [Time_t](#)

```
type Time_t int64
```

func [Time](#)

```
func Time(t *Time_t) (tt Time_t, err error)
```

type [Timespec](#)

```
type Timespec struct {  
    Sec  int64  
    Nsec int64  
}
```

func [NsecToTimespec](#)

```
func NsecToTimespec(nsec int64) (ts Timespec)
```

func (***Timespec**) [Nano](#)

```
func (ts *Timespec) Nano() int64
```

func (***Timespec**) [Unix](#)

```
func (ts *Timespec) Unix() (sec int64, nsec int64)
```

type [Timeval](#)

```
type Timeval struct {  
    Sec  int64  
    Usec int64  
}
```

func [NsecToTimeval](#)

```
func NsecToTimeval(nsec int64) (tv Timeval)
```

func (*Timeval) [Nano](#)

```
func (tv *Timeval) Nano() int64
```

func (*Timeval) [Unix](#)

```
func (tv *Timeval) Unix() (sec int64, nsec int64)
```

type Timex

```
type Timex struct {
    Modes      uint32
    Pad_cgo_0  [4]byte
    Offset     int64
    Freq       int64
    Maxerror   int64
    Esterror   int64
    Status     int32
    Pad_cgo_1  [4]byte
    Constant   int64
    Precision  int64
    Tolerance  int64
    Time       Timeval
    Tick       int64
    Ppsfreq   int64
    Jitter     int64
    Shift      int32
    Pad_cgo_2  [4]byte
    Stabil     int64
    Jitcnt     int64
    Calcnt     int64
    Errcnt     int64
    Stbcnt     int64
    Tai        int32
    Pad_cgo_3  [44]byte
}
```

type Tms

```
type Tms struct {  
    Utime  int64  
    Stime  int64  
    Ctime  int64  
    Cstime int64  
}
```

type [Ucred](#)

```
type Ucred struct {  
    Pid int32  
    Uid uint32  
    Gid uint32  
}
```

func [ParseUnixCredentials](#)

```
func ParseUnixCredentials(msg *SocketControlMessage) (*Ucred, error)
```

ParseUnixCredentials decodes a socket control message that contains credentials in a Ucred structure. To receive such a message, the SO_PASSCRED option must be enabled on the socket.

type Ustat_t

```
type Ustat_t struct {
    Tfree      int32
    Pad_cgo_0  [4]byte
    Tinode     uint64
    Fname      [6]int8
    Fpack      [6]int8
    Pad_cgo_1  [4]byte
}
```

type Utimbuf

```
type Utimbuf struct {  
    Actime  int64  
    Modtime int64  
}
```

type Utsname

```
type Utsname struct {
    Sysname    [65]int8
    Nodename   [65]int8
    Release    [65]int8
    Version    [65]int8
    Machine    [65]int8
    Domainname [65]int8
}
```

type [WaitStatus](#)

type WaitStatus uint32

func (WaitStatus) [Continued](#)

func (w WaitStatus) Continued() bool

func (WaitStatus) [CoreDump](#)

func (w WaitStatus) CoreDump() bool

func (WaitStatus) [ExitStatus](#)

func (w WaitStatus) ExitStatus() int

func (WaitStatus) [Exited](#)

func (w WaitStatus) Exited() bool

func (WaitStatus) [Signal](#)

func (w WaitStatus) Signal() Signal

func (WaitStatus) [Signaled](#)

func (w WaitStatus) Signaled() bool

func (WaitStatus) [StopSignal](#)

func (w WaitStatus) StopSignal() Signal

func (WaitStatus) [Stopped](#)

func (w WaitStatus) Stopped() bool

func (WaitStatus) [TrapCause](#)

```
func (w WaitStatus) TrapCause() int
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package testing

```
import "testing"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package testing provides support for automated testing of Go packages. It is intended to be used in concert with the “go test” command, which automates execution of any function of the form

```
func TestXxx(*testing.T)
```

where Xxx can be any alphanumeric string (but the first letter must not be in [a-z]) and serves to identify the test routine. These TestXxx routines should be declared within the package they are testing.

Functions of the form

```
func BenchmarkXxx(*testing.B)
```

are considered benchmarks, and are executed by the "go test" command when the -test.bench flag is provided.

A sample benchmark function looks like this:

```
func BenchmarkHello(b *testing.B) {  
    for i := 0; i < b.N; i++ {  
        fmt.Sprintf("hello")  
    }  
}
```

The benchmark package will vary b.N until the benchmark function lasts long enough to be timed reliably. The output

```
testing.BenchmarkHello    10000000    282 ns/op
```

means that the loop ran 10000000 times at a speed of 282 ns per loop.

If a benchmark needs some expensive setup before running, the timer may be stopped:

```
func BenchmarkBigLen(b *testing.B) {  
    b.StopTimer()  
    big := NewBig()  
    b.StartTimer()  
    for i := 0; i < b.N; i++ {
```

```

        big.Len()
    }
}

```

The package also runs and verifies example code. Example functions may include a concluding comment that begins with "Output:" and is compared with the standard output of the function when the tests are run, as in these examples of an example:

```

func ExampleHello() {
    fmt.Println("hello")
    // Output: hello
}

func ExampleSalutations() {
    fmt.Println("hello, and")
    fmt.Println("goodbye")
    // Output:
    // hello, and
    // goodbye
}

```

Example functions without output comments are compiled but not executed.

The naming convention to declare examples for a function F, a type T and method M on type T are:

```

func ExampleF() { ... }
func ExampleT() { ... }
func ExampleT_M() { ... }

```

Multiple example functions for a type/function/method may be provided by appending a distinct suffix to the name. The suffix must start with a lower-case letter.

```

func ExampleF_suffix() { ... }
func ExampleT_suffix() { ... }
func ExampleT_M_suffix() { ... }

```

The entire test file is presented as the example when it contains a single example function, at least one other function, type, variable, or constant declaration, and no test or benchmark functions.

Index

[func Main\(matchString func\(pat, str string\) \(bool, error\), tests \[\]InternalTest, benchmarks \[\]InternalBenchmark, examples \[\]InternalExample\)](#)
[func RunBenchmarks\(matchString func\(pat, str string\) \(bool, error\), benchmarks \[\]InternalBenchmark\)](#)
[func RunExamples\(matchString func\(pat, str string\) \(bool, error\), examples \[\]InternalExample\) \(ok bool\)](#)
[func RunTests\(matchString func\(pat, str string\) \(bool, error\), tests \[\]InternalTest\) \(ok bool\)](#)
[func Short\(\) bool](#)
type B
 [func \(c *B\) Error\(args ...interface{}\)](#)
 [func \(c *B\) Errorf\(format string, args ...interface{}\)](#)
 [func \(c *B\) Fail\(\)](#)
 [func \(c *B\) FailNow\(\)](#)
 [func \(c *B\) Failed\(\) bool](#)
 [func \(c *B\) Fatal\(args ...interface{}\)](#)
 [func \(c *B\) Fatalf\(format string, args ...interface{}\)](#)
 [func \(c *B\) Log\(args ...interface{}\)](#)
 [func \(c *B\) Logf\(format string, args ...interface{}\)](#)
 [func \(b *B\) ResetTimer\(\)](#)
 [func \(b *B\) SetBytes\(n int64\)](#)
 [func \(b *B\) StartTimer\(\)](#)
 [func \(b *B\) StopTimer\(\)](#)
type BenchmarkResult
 [func Benchmark\(f func\(b *B\)\) BenchmarkResult](#)
 [func \(r BenchmarkResult\) NsPerOp\(\) int64](#)
 [func \(r BenchmarkResult\) String\(\) string](#)
type InternalBenchmark
type InternalExample
type InternalTest
type T
 [func \(c *T\) Error\(args ...interface{}\)](#)
 [func \(c *T\) Errorf\(format string, args ...interface{}\)](#)
 [func \(c *T\) Fail\(\)](#)

[func \(c *T\) FailNow\(\)](#)
[func \(c *T\) Failed\(\) bool](#)
[func \(c *T\) Fatal\(args ...interface{}\)](#)
[func \(c *T\) Fatalf\(format string, args ...interface{}\)](#)
[func \(c *T\) Log\(args ...interface{}\)](#)
[func \(c *T\) Logf\(format string, args ...interface{}\)](#)
[func \(t *T\) Parallel\(\)](#)

Package files

[benchmark.go](#) [example.go](#) [testing.go](#)

func [Main](#)

```
func Main(matchString func(pat, str string) (bool, error), tests []I
```

An internal function but exported because it is cross-package; part of the implementation of the "go test" command.

func [RunBenchmarks](#)

```
func RunBenchmarks(matchString func(pat, str string) (bool, error),
```

An internal function but exported because it is cross-package; part of the implementation of the "go test" command.

func [RunExamples](#)

```
func RunExamples(matchString func(pat, str string) (bool, error), ex
```

func [RunTests](#)

```
func RunTests(matchString func(pat, str string) (bool, error), tests
```

func [Short](#)

```
func Short() bool
```

Short reports whether the `-test.short` flag is set.

type [B](#)

```
type B struct {  
    N int  
    // contains filtered or unexported fields  
}
```

B is a type passed to Benchmark functions to manage benchmark timing and to specify the number of iterations to run.

func (*B) [Error](#)

```
func (c *B) Error(args ...interface{})
```

Error is equivalent to Log() followed by Fail().

func (*B) [Errorf](#)

```
func (c *B) Errorf(format string, args ...interface{})
```

Errorf is equivalent to Logf() followed by Fail().

func (*B) [Fail](#)

```
func (c *B) Fail()
```

Fail marks the function as having failed but continues execution.

func (*B) [FailNow](#)

```
func (c *B) FailNow()
```

FailNow marks the function as having failed and stops its execution. Execution will continue at the next test or benchmark.

func (*B) [Failed](#)

```
func (c *B) Failed() bool
```

Failed returns whether the function has failed.

func (*B) [Fatal](#)

```
func (c *B) Fatal(args ...interface{})
```

Fatal is equivalent to Log() followed by FailNow().

func (*B) [Fatalf](#)

```
func (c *B) Fatalf(format string, args ...interface{})
```

Fatalf is equivalent to Logf() followed by FailNow().

func (*B) [Log](#)

```
func (c *B) Log(args ...interface{})
```

Log formats its arguments using default formatting, analogous to Println(), and records the text in the error log.

func (*B) [Logf](#)

```
func (c *B) Logf(format string, args ...interface{})
```

Logf formats its arguments according to the format, analogous to Printf(), and records the text in the error log.

func (*B) [ResetTimer](#)

```
func (b *B) ResetTimer()
```

ResetTimer sets the elapsed benchmark time to zero. It does not affect whether the timer is running.

func (*B) [SetBytes](#)

```
func (b *B) SetBytes(n int64)
```

SetBytes records the number of bytes processed in a single operation. If this is

called, the benchmark will report ns/op and MB/s.

func (*B) [StartTimer](#)

```
func (b *B) StartTimer()
```

StartTimer starts timing a test. This function is called automatically before a benchmark starts, but it can also be used to resume timing after a call to StopTimer.

func (*B) [StopTimer](#)

```
func (b *B) StopTimer()
```

StopTimer stops timing a test. This can be used to pause the timer while performing complex initialization that you don't want to measure.

type [BenchmarkResult](#)

```
type BenchmarkResult struct {  
    N      int           // The number of iterations.  
    T      time.Duration // The total time taken.  
    Bytes  int64           // Bytes processed in one iteration.  
}
```

The results of a benchmark run.

func [Benchmark](#)

```
func Benchmark(f func(b *B)) BenchmarkResult
```

Benchmark benchmarks a single function. Useful for creating custom benchmarks that do not use the "go test" command.

func (BenchmarkResult) [NsPerOp](#)

```
func (r BenchmarkResult) NsPerOp() int64
```

func (BenchmarkResult) [String](#)

```
func (r BenchmarkResult) String() string
```

type InternalBenchmark

```
type InternalBenchmark struct {  
    Name string  
    F     func(b *B)  
}
```

An internal type but exported because it is cross-package; part of the implementation of the "go test" command.

type [InternalExample](#)

```
type InternalExample struct {  
    Name    string  
    F       func()  
    Output  string  
}
```

type [InternalTest](#)

```
type InternalTest struct {  
    Name string  
    F     func(*T)  
}
```

An internal type but exported because it is cross-package; part of the implementation of the "go test" command.

type [T](#)

```
type T struct {  
    // contains filtered or unexported fields  
}
```

T is a type passed to Test functions to manage test state and support formatted test logs. Logs are accumulated during execution and dumped to standard error when done.

func (*T) [Error](#)

```
func (c *T) Error(args ...interface{})
```

Error is equivalent to Log() followed by Fail().

func (*T) [Errorf](#)

```
func (c *T) Errorf(format string, args ...interface{})
```

Errorf is equivalent to Logf() followed by Fail().

func (*T) [Fail](#)

```
func (c *T) Fail()
```

Fail marks the function as having failed but continues execution.

func (*T) [FailNow](#)

```
func (c *T) FailNow()
```

FailNow marks the function as having failed and stops its execution. Execution will continue at the next test or benchmark.

func (*T) [Failed](#)

```
func (c *T) Failed() bool
```

Failed returns whether the function has failed.

func (*T) [Fatal](#)

```
func (c *T) Fatal(args ...interface{})
```

Fatal is equivalent to Log() followed by FailNow().

func (*T) [Fatalf](#)

```
func (c *T) Fatalf(format string, args ...interface{})
```

Fatalf is equivalent to Logf() followed by FailNow().

func (*T) [Log](#)

```
func (c *T) Log(args ...interface{})
```

Log formats its arguments using default formatting, analogous to Println(), and records the text in the error log.

func (*T) [Logf](#)

```
func (c *T) Logf(format string, args ...interface{})
```

Logf formats its arguments according to the format, analogous to Printf(), and records the text in the error log.

func (*T) [Parallel](#)

```
func (t *T) Parallel()
```

Parallel signals that this test is to be run in parallel with (and only with) other parallel tests in this CPU group.

Subdirectories

Name **Synopsis**

[iotest](#) Package iotest implements Readers and Writers useful mainly for testing.

[quick](#) Package quick implements utility functions to help with black box testing.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package iotest

```
import "testing/iotest"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `iotest` implements Readers and Writers useful mainly for testing.

Index

Variables

[func DataErrReader\(r io.Reader\) io.Reader](#)

[func HalfReader\(r io.Reader\) io.Reader](#)

[func NewReadLogger\(prefix string, r io.Reader\) io.Reader](#)

[func NewWriteLogger\(prefix string, w io.Writer\) io.Writer](#)

[func OneByteReader\(r io.Reader\) io.Reader](#)

[func TimeoutReader\(r io.Reader\) io.Reader](#)

[func TruncateWriter\(w io.Writer, n int64\) io.Writer](#)

Package files

[logger.go](#) [reader.go](#) [writer.go](#)

Variables

```
var ErrTimeout = errors.New("timeout")
```

func [DataErrReader](#)

```
func DataErrReader(r io.Reader) io.Reader
```

DataErrReader returns a Reader that returns the final error with the last data read, instead of by itself with zero bytes of data.

func HalfReader

```
func HalfReader(r io.Reader) io.Reader
```

HalfReader returns a Reader that implements Read by reading half as many requested bytes from r.

func [NewReadLogger](#)

```
func NewReadLogger(prefix string, r io.Reader) io.Reader
```

NewReadLogger returns a reader that behaves like r except that it logs (using log.Print) each read to standard error, printing the prefix and the hexadecimal data written.

func [NewWriteLogger](#)

```
func NewWriteLogger(prefix string, w io.Writer) io.Writer
```

NewWriteLogger returns a writer that behaves like w except that it logs (using log.Printf) each write to standard error, printing the prefix and the hexadecimal data written.

func OneByteReader

```
func OneByteReader(r io.Reader) io.Reader
```

OneByteReader returns a Reader that implements each non-empty Read by reading one byte from r.

func TimeoutReader

```
func TimeoutReader(r io.Reader) io.Reader
```

TimeoutReader returns ErrTimeout on the second read with no data. Subsequent calls to read succeed.

func [TruncateWriter](#)

```
func TruncateWriter(w io.Writer, n int64) io.Writer
```

TruncateWriter returns a Writer that writes to w but stops silently after n bytes.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package quick

```
import "testing/quick"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package quick implements utility functions to help with black box testing.

Index

[func Check\(function interface{}, config *Config\) \(err error\)](#)
[func CheckEqual\(f, g interface{}, config *Config\) \(err error\)](#)
[func Value\(t reflect.Type, rand *rand.Rand\) \(value reflect.Value, ok bool\)](#)
[type CheckEqualError](#)
 [func \(s *CheckEqualError\) Error\(\) string](#)
[type CheckError](#)
 [func \(s *CheckError\) Error\(\) string](#)
[type Config](#)
[type Generator](#)
[type SetupError](#)
 [func \(s SetupError\) Error\(\) string](#)

Package files

quick.go

func [Check](#)

```
func Check(function interface{}, config *Config) (err error)
```

Check looks for an input to f, any function that returns bool, such that f returns false. It calls f repeatedly, with arbitrary values for each argument. If f returns false on a given input, Check returns that input as a *CheckError. For example:

```
func TestOddMultipleOfThree(t *testing.T) {
    f := func(x int) bool {
        y := OddMultipleOfThree(x)
        return y%2 == 1 && y%3 == 0
    }
    if err := quick.Check(f, nil); err != nil {
        t.Error(err)
    }
}
```

func [CheckEqual](#)

```
func CheckEqual(f, g interface{}, config *Config) (err error)
```

CheckEqual looks for an input on which f and g return different results. It calls f and g repeatedly with arbitrary values for each argument. If f and g return different answers, CheckEqual returns a *CheckEqualError describing the input and the outputs.

func [Value](#)

```
func Value(t reflect.Type, rand *rand.Rand) (value reflect.Value, ok
```

Value returns an arbitrary value of the given type. If the type implements the Generator interface, that will be used. Note: To create arbitrary values for structs, all the fields must be exported.

type [CheckEqualError](#)

```
type CheckEqualError struct {  
    CheckError  
    Out1 []interface{}  
    Out2 []interface{}  
}
```

A CheckEqualError is the result CheckEqual finding an error.

func (*CheckEqualError) [Error](#)

```
func (s *CheckEqualError) Error() string
```

type [CheckError](#)

```
type CheckError struct {  
    Count int  
    In    []interface{ }  
}
```

A CheckError is the result of Check finding an error.

func (*CheckError) [Error](#)

```
func (s *CheckError) Error() string
```

type [Config](#)

```
type Config struct {
    // MaxCount sets the maximum number of iterations. If zero,
    // MaxCountScale is used.
    MaxCount int
    // MaxCountScale is a non-negative scale factor applied to the d
    // maximum. If zero, the default is unchanged.
    MaxCountScale float64
    // If non-nil, rand is a source of random numbers. Otherwise a d
    // pseudo-random source will be used.
    Rand *rand.Rand
    // If non-nil, the Values function generates a slice of arbitrar
    // reflect.Values that are congruent with the arguments to the f
    // being tested. Otherwise, the top-level Values function is use
    // to generate them.
    Values func([]reflect.Value, *rand.Rand)
}
```

A Config structure contains options for running a test.

type Generator

```
type Generator interface {  
    // Generate returns a random instance of the type on which it is  
    // method using the size as a size hint.  
    Generate(rand *rand.Rand, size int) reflect.Value  
}
```

A Generator can generate random values of its own type.

type [SetupError](#)

```
type SetupError string
```

A SetupError is the result of an error in the way that check is being used, independent of the functions being tested.

func (SetupError) [Error](#)

```
func (s SetupError) Error() string
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Directory /src/pkg/text

Name	Synopsis
scanner	Package scanner provides a scanner and tokenizer for UTF-8-encoded text.
tabwriter	Package tabwriter implements a write filter (tabwriter.Writer) that translates tabbed columns in input into properly aligned text.
template	Package template implements data-driven templates for generating textual output.
parse	Package parse builds parse trees for templates as defined by text/template and html/template.

Need more packages? Take a look at the [Go Project Dashboard](#).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package scanner

```
import "text/scanner"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package scanner provides a scanner and tokenizer for UTF-8-encoded text. It takes an `io.Reader` providing the source, which then can be tokenized through repeated calls to the `Scan` function. For compatibility with existing tools, the NUL character is not allowed.

By default, a Scanner skips white space and Go comments and recognizes all literals as defined by the Go language specification. It may be customized to recognize only a subset of those literals and to recognize different white space characters.

Basic usage pattern:

```
var s scanner.Scanner
s.Init(src)
tok := s.Scan()
for tok != scanner.EOF {
    // do something with tok
    tok = s.Scan()
}
```

Index

Constants

[func TokenString\(tok rune\) string](#)

[type Position](#)

[func \(pos *Position\) IsValid\(\) bool](#)

[func \(pos Position\) String\(\) string](#)

[type Scanner](#)

[func \(s *Scanner\) Init\(src io.Reader\) *Scanner](#)

[func \(s *Scanner\) Next\(\) rune](#)

[func \(s *Scanner\) Peek\(\) rune](#)

[func \(s *Scanner\) Pos\(\) \(pos Position\)](#)

[func \(s *Scanner\) Scan\(\) rune](#)

[func \(s *Scanner\) TokenText\(\) string](#)

Package files

[scanner.go](#)

Constants

```
const (  
    ScanIdents      = 1 << -Ident  
    ScanInts        = 1 << -Int  
    ScanFloats      = 1 << -Float // includes Ints  
    ScanChars       = 1 << -Char  
    ScanStrings     = 1 << -String  
    ScanRawStrings  = 1 << -RawString  
    ScanComments    = 1 << -Comment  
    SkipComments    = 1 << -skipComment // if set with ScanComments,  
    GoTokens        = ScanIdents | ScanFloats | ScanChars | ScanStrin  
)
```

Predefined mode bits to control recognition of tokens. For instance, to configure a Scanner such that it only recognizes (Go) identifiers, integers, and skips comments, set the Scanner's Mode field to:

```
ScanIdents | ScanInts | SkipComments
```

```
const (  
    EOF = -(iota + 1)  
    Ident  
    Int  
    Float  
    Char  
    String  
    RawString  
    Comment  
)
```

The result of Scan is one of the following tokens or a Unicode character.

```
const GoWhitespace = 1<<'\t' | 1<<'\n' | 1<<'\r' | 1<<' '
```

GoWhitespace is the default value for the Scanner's Whitespace field. Its value selects Go's white space characters.

func [TokenString](#)

```
func TokenString(tok rune) string
```

TokenString returns a printable string for a token or Unicode character.

type [Position](#)

```
type Position struct {
    Filename string // filename, if any
    Offset   int    // byte offset, starting at 0
    Line     int    // line number, starting at 1
    Column   int    // column number, starting at 1 (character count)
}
```

A source position is represented by a Position value. A position is valid if Line > 0.

func (*Position) [IsValid](#)

```
func (pos *Position) IsValid() bool
```

IsValid returns true if the position is valid.

func (Position) [String](#)

```
func (pos Position) String() string
```

type [Scanner](#)

```
type Scanner struct {  
  
    // Error is called for each error encountered. If no Error  
    // function is set, the error is reported to os.Stderr.  
    Error func(s *Scanner, msg string)  
  
    // ErrorCount is incremented by one for each error encountered.  
    ErrorCount int  
  
    // The Mode field controls which tokens are recognized. For inst  
    // to recognize Ints, set the ScanInts bit in Mode. The field ma  
    // changed at any time.  
    Mode uint  
  
    // The Whitespace field controls which characters are recognized  
    // as white space. To recognize a character ch <= ' ' as white s  
    // set the ch'th bit in Whitespace (the Scanner's behavior is un  
    // for values ch > ' '). The field may be changed at any time.  
    Whitespace uint64  
  
    // Start position of most recently scanned token; set by Scan.  
    // Calling Init or Next invalidates the position (Line == 0).  
    // The Filename field is always left untouched by the Scanner.  
    // If an error is reported (via Error) and Position is invalid,  
    // the scanner is not inside a token. Call Pos to obtain an erro  
    // position in that case.  
    Position  
    // contains filtered or unexported fields  
}
```

A Scanner implements reading of Unicode characters and tokens from an io.Reader.

func (*Scanner) [Init](#)

```
func (s *Scanner) Init(src io.Reader) *Scanner
```

Init initializes a Scanner with a new source and returns s. Error is set to nil, ErrorCount is set to 0, Mode is set to GoTokens, and Whitespace is set to GoWhitespace.

func (*Scanner) [Next](#)

```
func (s *Scanner) Next() rune
```

Next reads and returns the next Unicode character. It returns EOF at the end of the source. It reports a read error by calling `s.Error`, if not nil; otherwise it prints an error message to `os.Stderr`. Next does not update the Scanner's Position field; use `Pos()` to get the current position.

func (*Scanner) [Peek](#)

```
func (s *Scanner) Peek() rune
```

Peek returns the next Unicode character in the source without advancing the scanner. It returns EOF if the scanner's position is at the last character of the source.

func (*Scanner) [Pos](#)

```
func (s *Scanner) Pos() (pos Position)
```

Pos returns the position of the character immediately after the character or token returned by the last call to `Next` or `Scan`.

func (*Scanner) [Scan](#)

```
func (s *Scanner) Scan() rune
```

Scan reads the next token or Unicode character from source and returns it. It only recognizes tokens `t` for which the respective Mode bit (`1<<-t`) is set. It returns EOF at the end of the source. It reports scanner errors (read and token errors) by calling `s.Error`, if not nil; otherwise it prints an error message to `os.Stderr`.

func (*Scanner) [TokenText](#)

```
func (s *Scanner) TokenText() string
```

TokenText returns the string corresponding to the most recently scanned token. Valid after calling `Scan()`.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package tabwriter

```
import "text/tabwriter"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package tabwriter implements a write filter (tabwriter.Writer) that translates tabbed columns in input into properly aligned text.

The package is using the Elastic Tabstops algorithm described at <http://nickgravgaard.com/elasticstops/index.html>.

Index

Constants

type Writer

func NewWriter(output io.Writer, minwidth, tabwidth, padding int, padchar byte, flags uint) *Writer

func (b *Writer) Flush() (err error)

func (b *Writer) Init(output io.Writer, minwidth, tabwidth, padding int, padchar byte, flags uint) *Writer

func (b *Writer) Write(buf []byte) (n int, err error)

Examples

Writer.Init

Package files

tabwriter.go

Constants

```
const (  
    // Ignore html tags and treat entities (starting with '&'   
    // and ending in ';') as single characters (width = 1).   
    FilterHTML uint = 1 << iota   
  
    // Strip Escape characters bracketing escaped text segments   
    // instead of passing them through unchanged with the text.   
    StripEscape   
  
    // Force right-alignment of cell content.   
    // Default is left-alignment.   
    AlignRight   
  
    // Handle empty columns as if they were not present in   
    // the input in the first place.   
    DiscardEmptyColumns   
  
    // Always use tabs for indentation columns (i.e., padding of   
    // leading empty cells on the left) independent of padchar.   
    TabIndent   
  
    // Print a vertical bar ('|') between columns (after formatting)   
    // Discarded columns appear as zero-width columns ("||").   
    Debug   
)
```

Formatting can be controlled with these flags.

```
const Escape = '\xff'
```

To escape a text segment, bracket it with Escape characters. For instance, the tab in this string "Ignore this tab: \xff\t\xff" does not terminate a cell and constitutes a single character of width one for formatting purposes.

The value 0xff was chosen because it cannot appear in a valid UTF-8 sequence.

type [Writer](#)

```
type Writer struct {  
    // contains filtered or unexported fields  
}
```

A `Writer` is a filter that inserts padding around tab-delimited columns in its input to align them in the output.

The `Writer` treats incoming bytes as UTF-8 encoded text consisting of cells terminated by (horizontal or vertical) tabs or line breaks (newline or formfeed characters). Cells in adjacent lines constitute a column. The `Writer` inserts padding as needed to make all cells in a column have the same width, effectively aligning the columns. It assumes that all characters have the same width except for tabs for which a `tabwidth` must be specified. Note that cells are tab-terminated, not tab-separated: trailing non-tab text at the end of a line does not form a column cell.

The `Writer` assumes that all Unicode code points have the same width; this may not be true in some fonts.

If `DiscardEmptyColumns` is set, empty columns that are terminated entirely by vertical (or "soft") tabs are discarded. Columns terminated by horizontal (or "hard") tabs are not affected by this flag.

If a `Writer` is configured to filter HTML, HTML tags and entities are passed through. The widths of tags and entities are assumed to be zero (tags) and one (entities) for formatting purposes.

A segment of text may be escaped by bracketing it with Escape characters. The `tabwriter` passes escaped text segments through unchanged. In particular, it does not interpret any tabs or line breaks within the segment. If the `StripEscape` flag is set, the Escape characters are stripped from the output; otherwise they are passed through as well. For the purpose of formatting, the width of the escaped text is always computed excluding the Escape characters.

The formfeed character (`\f`) acts like a newline but it also terminates all columns in the current line (effectively calling `Flush`). Cells in the next line start new columns. Unless found inside an HTML tag or inside an escaped text segment,

formfeed characters appear as newlines in the output.

The Writer must buffer input internally, because proper spacing of one line may depend on the cells in future lines. Clients must call Flush when done calling Write.

func [NewWriter](#)

```
func NewWriter(output io.Writer, minwidth, tabwidth, padding int, pa
```

NewWriter allocates and initializes a new tabwriter.Writer. The parameters are the same as for the the Init function.

func (*Writer) [Flush](#)

```
func (b *Writer) Flush() (err error)
```

Flush should be called after the last call to Write to ensure that any data buffered in the Writer is written to output. Any incomplete escape sequence at the end is considered complete for formatting purposes.

func (*Writer) [Init](#)

```
func (b *Writer) Init(output io.Writer, minwidth, tabwidth, padding
```

A Writer must be initialized with a call to Init. The first parameter (output) specifies the filter output. The remaining parameters control the formatting:

minwidth	minimal cell width including any padding
tabwidth	width of tab characters (equivalent number of spaces
padding	padding added to a cell before computing its width
padchar	ASCII char used for padding if padchar == '\t', the Writer will assume that the width of a '\t' in the formatted output is tabwidth, and cells are left-aligned independent of align_left (for correct-looking results, tabwidth must correspo
flags	formatting control

? Example

? Example

Code:

```
w := new(tabwriter.Writer)

// Format in tab-separated columns with a tab stop of 8.
w.Init(os.Stdout, 0, 8, 0, '\t', 0)
fmt.Fprintln(w, "a\tb\tc\td\t.")
fmt.Fprintln(w, "123\t12345\t1234567\t123456789\t.")
fmt.Fprintln(w)
w.Flush()

// Format right-aligned in space-separated columns of minimal width
// and at least one blank of padding (so wider column entries do not
// touch each other).
w.Init(os.Stdout, 5, 0, 1, ' ', tabwriter.AlignRight)
fmt.Fprintln(w, "a\tb\tc\td\t.")
fmt.Fprintln(w, "123\t12345\t1234567\t123456789\t.")
fmt.Fprintln(w)
w.Flush()
```

Output:

```
a      b      c      d      .
123    12345  1234567 123456789 .

      a      b      c      d.
    123 12345 1234567 123456789.
```

func (*Writer) [Write](#)

```
func (b *Writer) Write(buf []byte) (n int, err error)
```

Write writes buf to the writer b. The only errors returned are ones encountered while writing to the underlying output stream.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package template

```
import "text/template"
```

[Overview](#)

[Index](#)

[Examples](#)

[Subdirectories](#)

Overview ?

Overview ?

Package `template` implements data-driven templates for generating textual output.

To generate HTML output, see package `html/template`, which has the same interface as this package but automatically secures HTML output against certain attacks.

Templates are executed by applying them to a data structure. Annotations in the template refer to elements of the data structure (typically a field of a struct or a key in a map) to control execution and derive values to be displayed. Execution of the template walks the structure and sets the cursor, represented by a period '.' and called "dot", to the value at the current location in the structure as execution proceeds.

The input text for a template is UTF-8-encoded text in any format. "Actions"--data evaluations or control structures--are delimited by "{{" and "}}"; all text outside actions is copied to the output unchanged. Actions may not span newlines, although comments can.

Once constructed, a template may be executed safely in parallel.

Here is a trivial example that prints "17 items are made of wool".

```
type Inventory struct {
    Material string
    Count    uint
}
sweaters := Inventory{"wool", 17}
tmpl, err := template.New("test").Parse("{{.Count}} items are made of")
if err != nil { panic(err) }
err = tmpl.Execute(os.Stdout, sweaters)
if err != nil { panic(err) }
```

More intricate examples appear below.

Actions

Here is the list of actions. "Arguments" and "pipelines" are evaluations of data,

defined in detail below.

```
{{/* a comment */}}
```

A comment; discarded. May contain newlines.
Comments do not nest.

```
{{pipeline}}
```

The default textual representation of the value of the pipeline is copied to the output.

```
{{if pipeline}} T1 {{end}}
```

If the value of the pipeline is empty, no output is generated; otherwise, T1 is executed. The empty values are false, 0, a nil pointer or interface value, and any array, slice, map, or string of length zero.
Dot is unaffected.

```
{{if pipeline}} T1 {{else}} T0 {{end}}
```

If the value of the pipeline is empty, T0 is executed; otherwise, T1 is executed. Dot is unaffected.

```
{{range pipeline}} T1 {{end}}
```

The value of the pipeline must be an array, slice, or map. If the value of the pipeline has length zero, nothing is output; otherwise, dot is set to the successive elements of the array, slice, or map and T1 is executed. If the value is a map and keys are of basic type with a defined order ("comparable"), elements will be visited in sorted key order.

```
{{range pipeline}} T1 {{else}} T0 {{end}}
```

The value of the pipeline must be an array, slice, or map. If the value of the pipeline has length zero, dot is unaffected; T0 is executed; otherwise, dot is set to the successive elements of the array, slice, or map and T1 is executed.

```
{{template "name"}}
```

The template with the specified name is executed with nil as dot.

```
{{template "name" pipeline}}
```

The template with the specified name is executed with dot set to the value of the pipeline.

```
{{with pipeline}} T1 {{end}}
```

If the value of the pipeline is empty, no output is generated; otherwise, dot is set to the value of the pipeline and T1 is executed.

```
{{with pipeline}} T1 {{else}} T0 {{end}}
```

If the value of the pipeline is empty, dot is unaffected and T0 is executed; otherwise, dot is set to the value of the pipeline and T1 is executed.

is executed; otherwise, dot is set to the value of the pipel and T1 is executed.

Arguments

An argument is a simple value, denoted by one of the following.

- A boolean, string, character, integer, floating-point, imaginary or complex constant in Go syntax. These behave like Go's untyped constants, although raw strings may not span newlines.
- The character '.' (period):

`.`
The result is the value of dot.

- A variable name, which is a (possibly empty) alphanumeric string preceded by a dollar sign, such as
`$piOver2`

or

`$`

The result is the value of the variable.
Variables are described below.

- The name of a field of the data, which must be a struct, preceded by a period, such as
`.Field`

The result is the value of the field. Field invocations may be chained:

`.Field1.Field2`

Fields can also be evaluated on variables, including chaining:
`$x.Field1.Field2`

- The name of a key of the data, which must be a map, preceded by a period, such as
`.Key`

The result is the map element value indexed by the key.
Key invocations may be chained and combined with fields to any depth:

`.Field1.Key1.Field2.Key2`

Although the key must be an alphanumeric identifier, unlike with field names they do not need to start with an upper case letter.
Keys can also be evaluated on variables, including chaining:

`$x.key1.key2`

- The name of a niladic method of the data, preceded by a period, such as

`.Method`

The result is the value of invoking the method with dot as the receiver, `dot.Method()`. Such a method must have one return value (any type) or two return values, the second of which is an error. If it has two and the returned error is non-nil, execution terminates and an error is returned to the caller as the value of `Execute`. Method invocations may be chained and combined with fields and key

to any depth:

```
.Field1.Key1.Method1.Field2.Key2.Method2
```

Methods can also be evaluated on variables, including chaining:

```
$x.Method1.Field
```

- The name of a niladic function, such as
fun

The result is the value of invoking the function, fun(). The return types and values behave as in methods. Functions and function names are described below.

Arguments may evaluate to any type; if they are pointers the implementation automatically indirections to the base type when required. If an evaluation yields a function value, such as a function-valued field of a struct, the function is not invoked automatically, but it can be used as a truth value for an if action and the like. To invoke it, use the call function, defined below.

A pipeline is a possibly chained sequence of "commands". A command is a simple value (argument) or a function or method call, possibly with multiple arguments:

Argument

The result is the value of evaluating the argument.

```
.Method [Argument...]
```

The method can be alone or the last element of a chain but, unlike methods in the middle of a chain, it can take arguments. The result is the value of calling the method with the arguments:

```
dot.Method(Argument1, etc.)
```

```
functionName [Argument...]
```

The result is the value of calling the function associated with the name:

```
function(Argument1, etc.)
```

Functions and function names are described below.

Pipelines

A pipeline may be "chained" by separating a sequence of commands with pipeline characters '|'. In a chained pipeline, the result of the each command is passed as the last argument of the following command. The output of the final command in the pipeline is the value of the pipeline.

The output of a command will be either one value or two values, the second of which has type error. If that second value is present and evaluates to non-nil, execution terminates and the error is returned to the caller of Execute.

Variables

A pipeline inside an action may initialize a variable to capture the result. The initialization has syntax

```
$variable := pipeline
```

where `$variable` is the name of the variable. An action that declares a variable produces no output.

If a "range" action initializes a variable, the variable is set to the successive elements of the iteration. Also, a "range" may declare two variables, separated by a comma:

```
$index, $element := pipeline
```

in which case `$index` and `$element` are set to the successive values of the array/slice index or map key and element, respectively. Note that if there is only one variable, it is assigned the element; this is opposite to the convention in Go range clauses.

A variable's scope extends to the "end" action of the control structure ("if", "with", or "range") in which it is declared, or to the end of the template if there is no such control structure. A template invocation does not inherit variables from the point of its invocation.

When execution begins, `$` is set to the data argument passed to `Execute`, that is, to the starting value of `dot`.

Examples

Here are some example one-line templates demonstrating pipelines and variables. All produce the quoted word "output":

```
{{"\output\"}}  
    A string constant.  
{{`output`}}  
    A raw string constant.  
{{printf "%q" "output"}}  
    A function call.  
{{"output" | printf "%q"}}  
    A function call whose final argument comes from the previous
```

```

    command.
{{"put" | printf "%s%s" "out" | printf "%q"}}
    A more elaborate call.
{{"output" | printf "%s" | printf "%q"}}
    A longer chain.
{{with "output"}}{{printf "%q" .}}{{end}}
    A with action using dot.
{{with $x := "output" | printf "%q"}}{{{$x}}}}{{end}}
    A with action that creates and uses a variable.
{{with $x := "output"}}{{printf "%q" $x}}}}{{end}}
    A with action that uses the variable in another action.
{{with $x := "output"}}{{{$x | printf "%q"}}}}}}{{end}}
    The same, but pipelined.

```

Functions

During execution functions are found in two function maps: first in the template, then in the global function map. By default, no functions are defined in the template but the Funcs method can be used to add them.

Predefined global functions are named as follows.

and

Returns the boolean AND of its arguments by returning the first empty argument or the last argument, that is, "and x y" behaves as "if x then y else x". All the arguments are evaluated.

call

Returns the result of calling the first argument, which must be a function, with the remaining arguments as parameters. Thus "call .X.Y 1 2" is, in Go notation, dot.X.Y(1, 2) where Y is a func-valued field, map entry, or the like. The first argument must be the result of an evaluation that yields a value of function type (as distinct from a predefined function such as print). The function must return either one or two result values, the second of which is of type error. If the arguments don't match the function or the returned error value is non-nil, execution stops.

html

Returns the escaped HTML equivalent of the textual representation of its arguments.

index

Returns the result of indexing its first argument by the following arguments. Thus "index x 1 2 3" is, in Go syntax, x[1][2][3]. Each indexed item must be a map, slice, or array

js

Returns the escaped JavaScript equivalent of the textual

representation of its arguments.

`len` Returns the integer length of its argument.

`not` Returns the boolean negation of its single argument.

`or` Returns the boolean OR of its arguments by returning the first non-empty argument or the last argument, that is, "or x y" behaves as "if x then x else y". All the arguments are evaluated.

`print` An alias for `fmt.Sprint`

`printf` An alias for `fmt.Sprintf`

`println` An alias for `fmt.Sprintln`

`urlquery` Returns the escaped value of the textual representation of its arguments in a form suitable for embedding in a URL query

The boolean functions take any zero value to be false and a non-zero value to be true.

Associated templates

Each template is named by a string specified when it is created. Also, each template is associated with zero or more other templates that it may invoke by name; such associations are transitive and form a name space of templates.

A template may use a template invocation to instantiate another associated template; see the explanation of the "template" action above. The name must be that of a template associated with the template that contains the invocation.

Nested template definitions

When parsing a template, another template may be defined and associated with the template being parsed. Template definitions must appear at the top level of the template, much like global variables in a Go program.

The syntax of such definitions is to surround each template declaration with a "define" and "end" action.

The define action names the template being created by providing a string

constant. Here is a simple example:

```
`{{define "T1"}}ONE{{end}}
{{define "T2"}}TWO{{end}}
{{define "T3"}}{{template "T1"}} {{template "T2"}}{{end}}
{{template "T3"}}`
```

This defines two templates, T1 and T2, and a third T3 that invokes the other two when it is executed. Finally it invokes T3. If executed this template will produce the text

```
ONE TWO
```

By construction, a template may reside in only one association. If it's necessary to have a template addressable from multiple associations, the template definition must be parsed multiple times to create distinct *Template values, or must be copied with the Clone or AddParseTree method.

Parse may be called multiple times to assemble the various associated templates; see the ParseFiles and ParseGlob functions and methods for simple ways to parse related templates stored in files.

A template may be executed directly or through ExecuteTemplate, which executes an associated template identified by name. To invoke our example above, we might write,

```
err := tmpl.Execute(os.Stdout, "no data needed")
if err != nil {
    log.Fatalf("execution failed: %s", err)
}
```

or to invoke a particular template explicitly by name,

```
err := tmpl.ExecuteTemplate(os.Stdout, "T2", "no data needed")
if err != nil {
    log.Fatalf("execution failed: %s", err)
}
```

Index

[func HTMLEscape\(w io.Writer, b \[\]byte\)](#)
[func HTMLEscapeString\(s string\) string](#)
[func HTMLEscaper\(args ...interface{}\) string](#)
[func JSEscape\(w io.Writer, b \[\]byte\)](#)
[func JSEscapeString\(s string\) string](#)
[func JSEscaper\(args ...interface{}\) string](#)
[func URLQueryEscaper\(args ...interface{}\) string](#)
[type FuncMap](#)
[type Template](#)
 [func Must\(t *Template, err error\) *Template](#)
 [func New\(name string\) *Template](#)
 [func ParseFiles\(filenamees ...string\) \(*Template, error\)](#)
 [func ParseGlob\(pattern string\) \(*Template, error\)](#)
 [func \(t *Template\) AddParseTree\(name string, tree *parse.Tree\)](#)
 [\(*Template, error\)](#)
 [func \(t *Template\) Clone\(\) \(*Template, error\)](#)
 [func \(t *Template\) Delims\(left, right string\) *Template](#)
 [func \(t *Template\) Execute\(wr io.Writer, data interface{}\) \(err error\)](#)
 [func \(t *Template\) ExecuteTemplate\(wr io.Writer, name string, data](#)
 [interface{}\) error](#)
 [func \(t *Template\) Funcs\(funcMap FuncMap\) *Template](#)
 [func \(t *Template\) Lookup\(name string\) *Template](#)
 [func \(t *Template\) Name\(\) string](#)
 [func \(t *Template\) New\(name string\) *Template](#)
 [func \(t *Template\) Parse\(text string\) \(*Template, error\)](#)
 [func \(t *Template\) ParseFiles\(filenamees ...string\) \(*Template, error\)](#)
 [func \(t *Template\) ParseGlob\(pattern string\) \(*Template, error\)](#)
 [func \(t *Template\) Templates\(\) \[\]*Template](#)

Examples

[Template](#)
[Template \(Func\)](#)
[Template \(Glob\)](#)
[Template \(Helpers\)](#)

[Template \(Share\)](#)

Package files

[doc.go](#) [exec.go](#) [funcs.go](#) [helper.go](#) [template.go](#)

func [HTMLEscape](#)

```
func HTMLEscape(w io.Writer, b []byte)
```

HTMLEscape writes to w the escaped HTML equivalent of the plain text data b.

func [HTMLEscapeString](#)

```
func HTMLEscapeString(s string) string
```

HTMLEscapeString returns the escaped HTML equivalent of the plain text data s.

func [HTMLEscaper](#)

```
func HTMLEscaper(args ...interface{}) string
```

HTMLEscaper returns the escaped HTML equivalent of the textual representation of its arguments.

func [JSEscape](#)

```
func JSEscape(w io.Writer, b []byte)
```

JSEscape writes to w the escaped JavaScript equivalent of the plain text data b.

func [JSEscapeString](#)

```
func JSEscapeString(s string) string
```

JSEscapeString returns the escaped JavaScript equivalent of the plain text data s.

func [JSEscaper](#)

```
func JSEscaper(args ...interface{}) string
```

JSEscaper returns the escaped JavaScript equivalent of the textual representation of its arguments.

func [URLQueryEscaper](#)

```
func URLQueryEscaper(args ...interface{}) string
```

URLQueryEscaper returns the escaped value of the textual representation of its arguments in a form suitable for embedding in a URL query.

type [FuncMap](#)

```
type FuncMap map[string]interface{}
```

FuncMap is the type of the map defining the mapping from names to functions. Each function must have either a single return value, or two return values of which the second has type error. In that case, if the second (error) argument evaluates to non-nil during execution, execution terminates and Execute returns that error.

type [Template](#)

```
type Template struct {
    *parse.Tree
    // contains filtered or unexported fields
}
```

Template is the representation of a parsed template. The `*parse.Tree` field is exported only for use by `html/template` and should be treated as unexported by all other clients.

? Example

? Example

Code:

```
// Define a template.
const letter = `
Dear {{.Name}},
{{if .Attended}}
It was a pleasure to see you at the wedding.{{else}}
It is a shame you couldn't make it to the wedding.{{end}}
{{with .Gift}}Thank you for the lovely {{.}}.
{{end}}
Best wishes,
Josie
`

// Prepare some data to insert into the template.
type Recipient struct {
    Name, Gift string
    Attended bool
}
var recipients = []Recipient{
    {"Aunt Mildred", "bone china tea set", true},
    {"Uncle John", "moleskin pants", false},
    {"Cousin Rodney", "", false},
}

// Create a new template and parse the letter into it.
t := template.Must(template.New("letter").Parse(letter))

// Execute the template for each recipient.
for _, r := range recipients {
```

```
    err := t.Execute(os.Stdout, r)
    if err != nil {
        log.Println("executing template:", err)
    }
}
```

Output:

Dear Aunt Mildred,

It was a pleasure to see you at the wedding.
Thank you for the lovely bone china tea set.

Best wishes,
Josie

Dear Uncle John,

It is a shame you couldn't make it to the wedding.
Thank you for the lovely moleskin pants.

Best wishes,
Josie

Dear Cousin Rodney,

It is a shame you couldn't make it to the wedding.

Best wishes,
Josie

? Example (Func)

? Example (Func)

This example demonstrates a custom function to process template text. It installs the `strings.Title` function and uses it to Make Title Text Look Good In Our Template's Output.

Code:

```
// First we create a FuncMap with which to register the function.
funcMap := template.FuncMap{
    // The name "title" is what the function will be called in the t
    "title": strings.Title,
}
```

```

// A simple template definition to test our function.
// We print the input text several ways:
// - the original
// - title-cased
// - title-cased and then printed with %q
// - printed with %q and then title-cased.
const templateText = `
Input: {{printf "%q" .}}
Output 0: {{title .}}
Output 1: {{title . | printf "%q"}}
Output 2: {{printf "%q" . | title}}
`

// Create a template, add the function map, and parse the text.
tmpl, err := template.New("titleTest").Funcs(funcMap).Parse(templateText)
if err != nil {
    log.Fatalf("parsing: %s", err)
}

// Run the template to verify the output.
err = tmpl.Execute(os.Stdout, "the go programming language")
if err != nil {
    log.Fatalf("execution: %s", err)
}

```

Output:

```

Input: "the go programming language"
Output 0: The Go Programming Language
Output 1: "The Go Programming Language"
Output 2: "The Go Programming Language"

```

? Example (Glob)

? Example (Glob)

Here we demonstrate loading a set of templates from a directory.

Code:

```

// Here we create a temporary directory and populate it with our sam
// template definition files; usually the template files would alrea
// exist in some location known to the program.
dir := createTestDir([]templateFile{
    // T0.tmpl is a plain template file that just invokes T1.

```

```

    {"T0.tpl", `T0 invokes T1: ({{template "T1"}})`},
    // T1.tpl defines a template, T1 that invokes T2.
    {"T1.tpl", `{{define "T1"}}T1 invokes T2: ({{template "T2"}}){{
    // T2.tpl defines a template T2.
    {"T2.tpl", `{{define "T2"}}This is T2{{end}}`},
    })
// Clean up after the test; another quirk of running as an example.
defer os.RemoveAll(dir)

// pattern is the glob pattern used to find all the template files.
pattern := filepath.Join(dir, "*.tpl")

// Here starts the example proper.
// T0.tpl is the first name matched, so it becomes the starting tem
// the value returned by ParseGlob.
tpl := template.Must(template.ParseGlob(pattern))

err := tpl.Execute(os.Stdout, nil)
if err != nil {
    log.Fatalf("template execution: %s", err)
}

```

Output:

```
T0 invokes T1: (T1 invokes T2: (This is T2))
```

? Example (Helpers)

? Example (Helpers)

This example demonstrates one way to share some templates and use them in different contexts. In this variant we add multiple driver templates by hand to an existing bundle of templates.

Code:

```

// Here we create a temporary directory and populate it with our sam
// template definition files; usually the template files would alrea
// exist in some location known to the program.
dir := createTestDir([]templateFile{
    // T1.tpl defines a template, T1 that invokes T2.
    {"T1.tpl", `{{define "T1"}}T1 invokes T2: ({{template "T2"}}){{
    // T2.tpl defines a template T2.
    {"T2.tpl", `{{define "T2"}}This is T2{{end}}`},
    })
// Clean up after the test; another quirk of running as an example.
defer os.RemoveAll(dir)

```

```

// pattern is the glob pattern used to find all the template files.
pattern := filepath.Join(dir, "*.tmpl")

// Here starts the example proper.
// Load the helpers.
templates := template.Must(template.ParseGlob(pattern))
// Add one driver template to the bunch; we do this with an explicit
_, err := templates.Parse("{{define `driver1`}}Driver 1 calls T1: ({{
if err != nil {
    log.Fatal("parsing driver1: ", err)
}
// Add another driver template.
_, err = templates.Parse("{{define `driver2`}}Driver 2 calls T2: ({{
if err != nil {
    log.Fatal("parsing driver2: ", err)
}
// We load all the templates before execution. This package does not
// that behavior but html/template's escaping does, so it's a good h
err = templates.ExecuteTemplate(os.Stdout, "driver1", nil)
if err != nil {
    log.Fatalf("driver1 execution: %s", err)
}
err = templates.ExecuteTemplate(os.Stdout, "driver2", nil)
if err != nil {
    log.Fatalf("driver2 execution: %s", err)
}
}

```

Output:

```

Driver 1 calls T1: (T1 invokes T2: (This is T2))
Driver 2 calls T2: (This is T2)

```

? Example (Share)

? Example (Share)

This example demonstrates how to use one group of driver templates with distinct sets of helper templates.

Code:

```

// Here we create a temporary directory and populate it with our sam
// template definition files; usually the template files would alrea
// exist in some location known to the program.
dir := createTestDir([]templateFile{
    // T0.tmpl is a plain template file that just invokes T1.

```

```

    {"T0.tpl", "T0 ({{.}} version) invokes T1: ({{template `T1`}})\
    // T1.tpl defines a template, T1 that invokes T2. Note T2 is no
    {"T1.tpl", `{{define "T1"}}T1 invokes T2: ({{template "T2"}}){{
}})
// Clean up after the test; another quirk of running as an example.
defer os.RemoveAll(dir)

// pattern is the glob pattern used to find all the template files.
pattern := filepath.Join(dir, "*.tpl")

// Here starts the example proper.
// Load the drivers.
drivers := template.Must(template.ParseGlob(pattern))

// We must define an implementation of the T2 template. First we clo
// the drivers, then add a definition of T2 to the template name spa

// 1. Clone the helper set to create a new name space from which to
first, err := drivers.Clone()
if err != nil {
    log.Fatal("cloning helpers: ", err)
}
// 2. Define T2, version A, and parse it.
_, err = first.Parse("{{define `T2`}}T2, version A{{end}}")
if err != nil {
    log.Fatal("parsing T2: ", err)
}

// Now repeat the whole thing, using a different version of T2.
// 1. Clone the drivers.
second, err := drivers.Clone()
if err != nil {
    log.Fatal("cloning drivers: ", err)
}
// 2. Define T2, version B, and parse it.
_, err = second.Parse("{{define `T2`}}T2, version B{{end}}")
if err != nil {
    log.Fatal("parsing T2: ", err)
}

// Execute the templates in the reverse order to verify the
// first is unaffected by the second.
err = second.ExecuteTemplate(os.Stdout, "T0.tpl", "second")
if err != nil {
    log.Fatalf("second execution: %s", err)
}
err = first.ExecuteTemplate(os.Stdout, "T0.tpl", "first")
if err != nil {
    log.Fatalf("first: execution: %s", err)
}

```

Output:

```
T0 (second version) invokes T1: (T1 invokes T2: (T2, version B))
T0 (first version) invokes T1: (T1 invokes T2: (T2, version A))
```

func [Must](#)

```
func Must(t *Template, err error) *Template
```

Must is a helper that wraps a call to a function returning (*Template, error) and panics if the error is non-nil. It is intended for use in variable initializations such as

```
var t = template.Must(template.New("name").Parse("text"))
```

func [New](#)

```
func New(name string) *Template
```

New allocates a new template with the given name.

func [ParseFiles](#)

```
func ParseFiles(filenamees ...string) (*Template, error)
```

ParseFiles creates a new Template and parses the template definitions from the named files. The returned template's name will have the (base) name and (parsed) contents of the first file. There must be at least one file. If an error occurs, parsing stops and the returned *Template is nil.

func [ParseGlob](#)

```
func ParseGlob(pattern string) (*Template, error)
```

ParseGlob creates a new Template and parses the template definitions from the files identified by the pattern, which must match at least one file. The returned template will have the (base) name and (parsed) contents of the first file matched by the pattern. ParseGlob is equivalent to calling ParseFiles with the list of files matched by the pattern.

func (*Template) [AddParseTree](#)

```
func (t *Template) AddParseTree(name string, tree *parse.Tree) (*Tem
```

AddParseTree creates a new template with the name and parse tree and associates it with t.

func (*Template) [Clone](#)

```
func (t *Template) Clone() (*Template, error)
```

Clone returns a duplicate of the template, including all associated templates. The actual representation is not copied, but the name space of associated templates is, so further calls to Parse in the copy will add templates to the copy but not to the original. Clone can be used to prepare common templates and use them with variant definitions for other templates by adding the variants after the clone is made.

func (*Template) [Delims](#)

```
func (t *Template) Delims(left, right string) *Template
```

Delims sets the action delimiters to the specified strings, to be used in subsequent calls to Parse, ParseFiles, or ParseGlob. Nested template definitions will inherit the settings. An empty delimiter stands for the corresponding default: {{ or }}. The return value is the template, so calls can be chained.

func (*Template) [Execute](#)

```
func (t *Template) Execute(wr io.Writer, data interface{}) (err error)
```

Execute applies a parsed template to the specified data object, and writes the output to wr.

func (*Template) [ExecuteTemplate](#)

```
func (t *Template) ExecuteTemplate(wr io.Writer, name string, data i
```

ExecuteTemplate applies the template associated with t that has the given name to the specified data object and writes the output to wr.

func (*Template) [Funcs](#)

```
func (t *Template) Funcs(funcMap FuncMap) *Template
```

Funcs adds the elements of the argument map to the template's function map. It panics if a value in the map is not a function with appropriate return type. However, it is legal to overwrite elements of the map. The return value is the template, so calls can be chained.

func (*Template) [Lookup](#)

```
func (t *Template) Lookup(name string) *Template
```

Lookup returns the template with the given name that is associated with t, or nil if there is no such template.

func (*Template) [Name](#)

```
func (t *Template) Name() string
```

Name returns the name of the template.

func (*Template) [New](#)

```
func (t *Template) New(name string) *Template
```

New allocates a new template associated with the given one and with the same delimiters. The association, which is transitive, allows one template to invoke another with a `{{template}}` action.

func (*Template) [Parse](#)

```
func (t *Template) Parse(text string) (*Template, error)
```

Parse parses a string into a template. Nested template definitions will be associated with the top-level template t. Parse may be called multiple times to parse definitions of templates to associate with t. It is an error if a resulting template is non-empty (contains content other than template definitions) and would replace a non-empty template with the same name. (In multiple calls to Parse with the same receiver template, only one call can contain text other than

space, comments, and template definitions.)

func (*Template) [ParseFiles](#)

```
func (t *Template) ParseFiles(filenamees ...string) (*Template, error)
```

ParseFiles parses the named files and associates the resulting templates with t. If an error occurs, parsing stops and the returned template is nil; otherwise it is t. There must be at least one file.

func (*Template) [ParseGlob](#)

```
func (t *Template) ParseGlob(pattern string) (*Template, error)
```

ParseGlob parses the template definitions in the files identified by the pattern and associates the resulting templates with t. The pattern is processed by filepath.Glob and must match at least one file. ParseGlob is equivalent to calling t.ParseFiles with the list of files matched by the pattern.

func (*Template) [Templates](#)

```
func (t *Template) Templates() []*Template
```

Templates returns a slice of the templates associated with t, including t itself.

Subdirectories

Name **Synopsis**

[parse](#) Package parse builds parse trees for templates as defined by text/template and html/template.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package parse

```
import "text/template/parse"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package parse builds parse trees for templates as defined by text/template and html/template. Clients should use those packages to construct templates rather than this one, which provides shared internal data structures not intended for general use.

Index

[func IsEmptyTree\(n Node\) bool](#)
[func Parse\(name, text, leftDelim, rightDelim string, funcs ...map\[string\]interface{ }\) \(treeSet map\[string\]*Tree, err error\)](#)
[type ActionNode](#)
 [func \(a *ActionNode\) Copy\(\) Node](#)
 [func \(a *ActionNode\) String\(\) string](#)
[type BoolNode](#)
 [func \(b *BoolNode\) Copy\(\) Node](#)
 [func \(b *BoolNode\) String\(\) string](#)
[type BranchNode](#)
 [func \(b *BranchNode\) String\(\) string](#)
[type CommandNode](#)
 [func \(c *CommandNode\) Copy\(\) Node](#)
 [func \(c *CommandNode\) String\(\) string](#)
[type DotNode](#)
 [func \(d *DotNode\) Copy\(\) Node](#)
 [func \(d *DotNode\) String\(\) string](#)
 [func \(d *DotNode\) Type\(\) NodeType](#)
[type FieldNode](#)
 [func \(f *FieldNode\) Copy\(\) Node](#)
 [func \(f *FieldNode\) String\(\) string](#)
[type IdentifierNode](#)
 [func NewIdentifier\(ident string\) *IdentifierNode](#)
 [func \(i *IdentifierNode\) Copy\(\) Node](#)
 [func \(i *IdentifierNode\) String\(\) string](#)
[type IfNode](#)
 [func \(i *IfNode\) Copy\(\) Node](#)
[type ListNode](#)
 [func \(l *ListNode\) Copy\(\) Node](#)
 [func \(l *ListNode\) CopyList\(\) *ListNode](#)
 [func \(l *ListNode\) String\(\) string](#)
[type Node](#)
[type NodeType](#)
 [func \(t NodeType\) Type\(\) NodeType](#)
[type NumberNode](#)

```
func (n *NumberNode) Copy() Node  
func (n *NumberNode) String() string  
type PipeNode  
func (p *PipeNode) Copy() Node  
func (p *PipeNode) CopyPipe() *PipeNode  
func (p *PipeNode) String() string  
type RangeNode  
func (r *RangeNode) Copy() Node  
type StringNode  
func (s *StringNode) Copy() Node  
func (s *StringNode) String() string  
type TemplateNode  
func (t *TemplateNode) Copy() Node  
func (t *TemplateNode) String() string  
type TextNode  
func (t *TextNode) Copy() Node  
func (t *TextNode) String() string  
type Tree  
func New(name string, funcs ...map[string]interface{ }) *Tree  
func (t *Tree) Parse(s, leftDelim, rightDelim string, treeSet  
map[string]*Tree, funcs ...map[string]interface{ }) (tree *Tree, err error)  
type VariableNode  
func (v *VariableNode) Copy() Node  
func (v *VariableNode) String() string  
type WithNode  
func (w *WithNode) Copy() Node
```

Package files

[lex.go](#) [node.go](#) [parse.go](#)

func IsEmptyTree

```
func IsEmptyTree(n Node) bool
```

IsEmptyTree reports whether this tree (node) is empty of everything but space.

func [Parse](#)

```
func Parse(name, text, leftDelim, rightDelim string, funcs ...map[st
```

Parse returns a map from template name to `parse.Tree`, created by parsing the templates described in the argument string. The top-level template will be given the specified name. If an error is encountered, parsing stops and an empty map is returned with the error.

type [ActionNode](#)

```
type ActionNode struct {  
    NodeType  
    Line int          // The line number in the input.  
    Pipe *PipeNode // The pipeline in the action.  
}
```

ActionNode holds an action (something bounded by delimiters). Control actions have their own nodes; ActionNode represents simple ones such as field evaluations.

func (*ActionNode) [Copy](#)

```
func (a *ActionNode) Copy() Node
```

func (*ActionNode) [String](#)

```
func (a *ActionNode) String() string
```

type [BoolNode](#)

```
type BoolNode struct {  
    NodeType  
    True bool // The value of the boolean constant.  
}
```

BoolNode holds a boolean constant.

func (***BoolNode**) [Copy](#)

```
func (b *BoolNode) Copy() Node
```

func (***BoolNode**) [String](#)

```
func (b *BoolNode) String() string
```

type [BranchNode](#)

```
type BranchNode struct {
    NodeType
    Line      int          // The line number in the input.
    Pipe      *PipeNode // The pipeline to be evaluated.
    List      *ListNode // What to execute if the value is non-empty.
    ElseList  *ListNode // What to execute if the value is empty (nil)
}
```

BranchNode is the common representation of if, range, and with.

func (*BranchNode) [String](#)

```
func (b *BranchNode) String() string
```

type [CommandNode](#)

```
type CommandNode struct {  
    NodeType  
    Args []Node // Arguments in lexical order: Identifier, field, or  
}
```

CommandNode holds a command (a pipeline inside an evaluating action).

func (*CommandNode) [Copy](#)

```
func (c *CommandNode) Copy() Node
```

func (*CommandNode) [String](#)

```
func (c *CommandNode) String() string
```

type [DotNode](#)

```
type DotNode bool
```

DotNode holds the special identifier '!'. It is represented by a nil pointer.

func (*DotNode) [Copy](#)

```
func (d *DotNode) Copy() Node
```

func (*DotNode) [String](#)

```
func (d *DotNode) String() string
```

func (*DotNode) [Type](#)

```
func (d *DotNode) Type() NodeType
```

type [FieldNode](#)

```
type FieldNode struct {  
    NodeType  
    Ident []string // The identifiers in lexical order.  
}
```

FieldNode holds a field (identifier starting with '.'). The names may be chained ('.x.y'). The period is dropped from each ident.

func (***FieldNode**) [Copy](#)

```
func (f *FieldNode) Copy() Node
```

func (***FieldNode**) [String](#)

```
func (f *FieldNode) String() string
```

type [IdentifierNode](#)

```
type IdentifierNode struct {  
    NodeType  
    Ident string // The identifier's name.  
}
```

IdentifierNode holds an identifier.

func [NewIdentifier](#)

```
func NewIdentifier(ident string) *IdentifierNode
```

NewIdentifier returns a new IdentifierNode with the given identifier name.

func (*IdentifierNode) [Copy](#)

```
func (i *IdentifierNode) Copy() Node
```

func (*IdentifierNode) [String](#)

```
func (i *IdentifierNode) String() string
```

type [IfNode](#)

```
type IfNode struct {  
    BranchNode  
}
```

IfNode represents an `{{if}}` action and its commands.

func (*IfNode) [Copy](#)

```
func (i *IfNode) Copy() Node
```

type [ListNode](#)

```
type ListNode struct {  
    NodeType  
    Nodes []Node // The element nodes in lexical order.  
}
```

ListNode holds a sequence of nodes.

func (*ListNode) [Copy](#)

```
func (l *ListNode) Copy() Node
```

func (*ListNode) [CopyList](#)

```
func (l *ListNode) CopyList() *ListNode
```

func (*ListNode) [String](#)

```
func (l *ListNode) String() string
```

type [Node](#)

```
type Node interface {
    Type() NodeType
    String() string
    // Copy does a deep copy of the Node and all its components.
    // To avoid type assertions, some XxxNodes also have specialized
    // CopyXxx methods that return *XxxNode.
    Copy() Node
}
```

A node is an element in the parse tree. The interface is trivial.

type [NodeType](#)

```
type NodeType int
```

NodeType identifies the type of a parse tree node.

```
const (  
    NodeText      NodeType = iota // Plain text.  
    NodeAction    // A simple action such as field eval  
    NodeBool      // A boolean constant.  
    NodeCommand   // An element of a pipeline.  
    NodeDot       // The cursor, dot.  
  
    NodeField     // A field or method name.  
    NodeIdentifier // An identifier; always a function name.  
    NodeIf        // An if action.  
    NodeList      // A list of Nodes.  
    NodeNumber    // A numerical constant.  
    NodePipe      // A pipeline of commands.  
    NodeRange     // A range action.  
    NodeString    // A string constant.  
    NodeTemplate  // A template invocation action.  
    NodeVariable  // A $ variable.  
    NodeWith      // A with action.  
)
```

func (NodeType) [Type](#)

```
func (t NodeType) Type() NodeType
```

Type returns itself and provides an easy default implementation for embedding in a Node. Embedded in all non-trivial Nodes.

type [NumberNode](#)

```
type NumberNode struct {
    NodeType
    IsInt      bool        // Number has an integral value.
    IsUint     bool        // Number has an unsigned integral value.
    IsFloat    bool        // Number has a floating-point value.
    IsComplex  bool        // Number is complex.
    Int64      int64       // The signed integer value.
    Uint64     uint64      // The unsigned integer value.
    Float64    float64    // The floating-point value.
    Complex128 complex128 // The complex value.
    Text       string      // The original textual representation fro
}
```

NumberNode holds a number: signed or unsigned integer, float, or complex. The value is parsed and stored under all the types that can represent the value. This simulates in a small amount of code the behavior of Go's ideal constants.

func (*NumberNode) [Copy](#)

```
func (n *NumberNode) Copy() Node
```

func (*NumberNode) [String](#)

```
func (n *NumberNode) String() string
```

type [PipeNode](#)

```
type PipeNode struct {
    NodeType
    Line int           // The line number in the input.
    Decl []*VariableNode // Variable declarations in lexical order.
    Cmds []*CommandNode // The commands in lexical order.
}
```

PipeNode holds a pipeline with optional declaration

func (*PipeNode) [Copy](#)

```
func (p *PipeNode) Copy() Node
```

func (*PipeNode) [CopyPipe](#)

```
func (p *PipeNode) CopyPipe() *PipeNode
```

func (*PipeNode) [String](#)

```
func (p *PipeNode) String() string
```

type [RangeNode](#)

```
type RangeNode struct {  
    BranchNode  
}
```

RangeNode represents a {{range}} action and its commands.

func (*RangeNode) [Copy](#)

```
func (r *RangeNode) Copy() Node
```

type [StringNode](#)

```
type StringNode struct {  
    NodeType  
    Quoted string // The original text of the string, with quotes.  
    Text    string // The string, after quote processing.  
}
```

StringNode holds a string constant. The value has been "unquoted".

func (*StringNode) [Copy](#)

```
func (s *StringNode) Copy() Node
```

func (*StringNode) [String](#)

```
func (s *StringNode) String() string
```

type [TemplateNode](#)

```
type TemplateNode struct {
    NodeType
    Line int        // The line number in the input.
    Name string     // The name of the template (unquoted).
    Pipe *PipeNode // The command to evaluate as dot for the templat
}
```

TemplateNode represents a {{template}} action.

func (*TemplateNode) [Copy](#)

```
func (t *TemplateNode) Copy() Node
```

func (*TemplateNode) [String](#)

```
func (t *TemplateNode) String() string
```

type [TextNode](#)

```
type TextNode struct {  
    NodeType  
    Text []byte // The text; may span newlines.  
}
```

TextNode holds plain text.

func (*TextNode) [Copy](#)

```
func (t *TextNode) Copy() Node
```

func (*TextNode) [String](#)

```
func (t *TextNode) String() string
```

type [Tree](#)

```
type Tree struct {
    Name string    // name of the template represented by the tree.
    Root *ListNode // top-level root of the tree.
    // Parsing only; cleared after parse.
    // contains filtered or unexported fields
}
```

Tree is the representation of a single parsed template.

func [New](#)

```
func New(name string, funcs ...map[string]interface{}) *Tree
```

New allocates a new parse tree with the given name.

func (***Tree**) [Parse](#)

```
func (t *Tree) Parse(s, leftDelim, rightDelim string, treeSet map[st
```

Parse parses the template definition string to construct a representation of the template for execution. If either action delimiter string is empty, the default ("{" or "}") is used. Embedded template definitions are added to the treeSet map.

type [VariableNode](#)

```
type VariableNode struct {  
    NodeType  
    Ident []string // Variable names in lexical order.  
}
```

VariableNode holds a list of variable names. The dollar sign is part of the name.

func (*VariableNode) [Copy](#)

```
func (v *VariableNode) Copy() Node
```

func (*VariableNode) [String](#)

```
func (v *VariableNode) String() string
```

type [WithNode](#)

```
type WithNode struct {  
    BranchNode  
}
```

WithNode represents a `{{with}}` action and its commands.

func (*WithNode) [Copy](#)

```
func (w *WithNode) Copy() Node
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package time

```
import "time"
```

[Overview](#)

[Index](#)

[Examples](#)

Overview ?

Overview ?

Package time provides functionality for measuring and displaying time.

The calendrical calculations always assume a Gregorian calendar.

Index

Constants

func After(d Duration) <-chan Time

func Sleep(d Duration)

func Tick(d Duration) <-chan Time

type Duration

func ParseDuration(s string) (Duration, error)

func Since(t Time) Duration

func (d Duration) Hours() float64

func (d Duration) Minutes() float64

func (d Duration) Nanoseconds() int64

func (d Duration) Seconds() float64

func (d Duration) String() string

type Location

func FixedZone(name string, offset int) *Location

func LoadLocation(name string) (*Location, error)

func (l *Location) String() string

type Month

func (m Month) String() string

type ParseError

func (e *ParseError) Error() string

type Ticker

func NewTicker(d Duration) *Ticker

func (t *Ticker) Stop()

type Time

func Date(year int, month Month, day, hour, min, sec, nsec int, loc *Location) Time

func Now() Time

func Parse(layout, value string) (Time, error)

func Unix(sec int64, nsec int64) Time

func (t Time) Add(d Duration) Time

func (t Time) AddDate(years int, months int, days int) Time

func (t Time) After(u Time) bool

func (t Time) Before(u Time) bool

func (t Time) Clock() (hour, min, sec int)

func (t Time) Date() (year int, month Month, day int)

[func \(t Time\) Day\(\) int](#)
[func \(t Time\) Equal\(u Time\) bool](#)
[func \(t Time\) Format\(layout string\) string](#)
[func \(t *Time\) GobDecode\(buf \[\]byte\) error](#)
[func \(t Time\) GobEncode\(\) \(\[\]byte, error\)](#)
[func \(t Time\) Hour\(\) int](#)
[func \(t Time\) ISOWeek\(\) \(year, week int\)](#)
[func \(t Time\) In\(loc *Location\) Time](#)
[func \(t Time\) IsZero\(\) bool](#)
[func \(t Time\) Local\(\) Time](#)
[func \(t Time\) Location\(\) *Location](#)
[func \(t Time\) MarshalJSON\(\) \(\[\]byte, error\)](#)
[func \(t Time\) Minute\(\) int](#)
[func \(t Time\) Month\(\) Month](#)
[func \(t Time\) Nanosecond\(\) int](#)
[func \(t Time\) Second\(\) int](#)
[func \(t Time\) String\(\) string](#)
[func \(t Time\) Sub\(u Time\) Duration](#)
[func \(t Time\) UTC\(\) Time](#)
[func \(t Time\) Unix\(\) int64](#)
[func \(t Time\) UnixNano\(\) int64](#)
[func \(t *Time\) UnmarshalJSON\(data \[\]byte\) \(err error\)](#)
[func \(t Time\) Weekday\(\) Weekday](#)
[func \(t Time\) Year\(\) int](#)
[func \(t Time\) Zone\(\) \(name string, offset int\)](#)

[type Timer](#)
[func AfterFunc\(d Duration, f func\(\)\) *Timer](#)
[func NewTimer\(d Duration\) *Timer](#)
[func \(t *Timer\) Stop\(\) \(ok bool\)](#)

[type Weekday](#)
[func \(d Weekday\) String\(\) string](#)

Examples

[After](#)

[Date](#)

[Duration](#)

[Month](#)

[Sleep](#)
[Tick](#)

Package files

[format.go](#) [sleep.go](#) [sys_unix.go](#) [tick.go](#) [time.go](#) [zoneinfo.go](#) [zoneinfo_read.go](#) [zoneinfo_unix.go](#)

Constants

```
const (  
  ANSIC           = "Mon Jan _2 15:04:05 2006"  
  UnixDate       = "Mon Jan _2 15:04:05 MST 2006"  
  RubyDate       = "Mon Jan 02 15:04:05 -0700 2006"  
  RFC822         = "02 Jan 06 15:04 MST"  
  RFC822Z        = "02 Jan 06 15:04 -0700" // RFC822 with numeric zon  
  RFC850         = "Monday, 02-Jan-06 15:04:05 MST"  
  RFC1123        = "Mon, 02 Jan 2006 15:04:05 MST"  
  RFC1123Z       = "Mon, 02 Jan 2006 15:04:05 -0700" // RFC1123 with  
  RFC3339        = "2006-01-02T15:04:05Z07:00"  
  RFC3339Nano    = "2006-01-02T15:04:05.999999999Z07:00"  
  Kitchen        = "3:04PM"  
  // Handy time stamps.  
  Stamp          = "Jan _2 15:04:05"  
  StampMilli     = "Jan _2 15:04:05.000"  
  StampMicro     = "Jan _2 15:04:05.000000"  
  StampNano      = "Jan _2 15:04:05.000000000"  
)
```

These are predefined layouts for use in `Time.Format`. The standard time used in the layouts is:

```
Mon Jan 2 15:04:05 MST 2006
```

which is Unix time 1136243045. Since MST is GMT-0700, the standard time can be thought of as

```
01/02 03:04:05PM '06 -0700
```

To define your own format, write down what the standard time would look like formatted your way; see the values of constants like `ANSIC`, `StampMicro` or `Kitchen` for examples.

Within the format string, an underscore `_` represents a space that may be replaced by a digit if the following number (a day) has two digits; for compatibility with fixed-width Unix time formats.

A decimal point followed by one or more zeros represents a fractional second, printed to the given number of decimal places. A decimal point followed by one or more nines represents a fractional second, printed to the given number of

decimal places, with trailing zeros removed. When parsing (only), the input may contain a fractional second field immediately after the seconds field, even if the layout does not signify its presence. In that case a decimal point followed by a maximal series of digits is parsed as a fractional second.

Numeric time zone offsets format as follows:

-0700 hhmm
-07:00 hh:mm

Replacing the sign in the format with a Z triggers the ISO 8601 behavior of printing Z instead of an offset for the UTC zone. Thus:

Z0700 Z or hhmm
Z07:00 Z or hh:mm

func [After](#)

```
func After(d Duration) <-chan Time
```

After waits for the duration to elapse and then sends the current time on the returned channel. It is equivalent to `NewTimer(d).C`.

? Example

? Example

Code:

```
select {
case m := <-c:
    handle(m)
case <-time.After(5 * time.Minute):
    fmt.Println("timed out")
}
```

func [Sleep](#)

```
func Sleep(d Duration)
```

Sleep pauses the current goroutine for the duration d.

? Example

? Example

Code:

```
time.Sleep(100 * time.Millisecond)
```

func [Tick](#)

```
func Tick(d Duration) <-chan Time
```

Tick is a convenience wrapper for NewTicker providing access to the ticking channel only. Useful for clients that have no need to shut down the ticker.

? Example

? Example

Code:

```
c := time.Tick(1 * time.Minute)
for now := range c {
    fmt.Printf("%v %s\n", now, statusUpdate())
}
```

type [Duration](#)

```
type Duration int64
```

A Duration represents the elapsed time between two instants as an int64 nanosecond count. The representation limits the largest representable duration to approximately 290 years.

```
const (  
    Nanosecond Duration = 1  
    Microsecond      = 1000 * Nanosecond  
    Millisecond       = 1000 * Microsecond  
    Second            = 1000 * Millisecond  
    Minute            = 60 * Second  
    Hour              = 60 * Minute  
)
```

Common durations. There is no definition for units of Day or larger to avoid confusion across daylight savings time zone transitions.

To count the number of units in a Duration, divide:

```
second := time.Second  
fmt.Print(int64(second/time.Millisecond)) // prints 1000
```

To convert an integer number of units to a Duration, multiply:

```
seconds := 10  
fmt.Print(time.Duration(seconds)*time.Second) // prints 10s
```

? Example

? Example

Code:

```
t0 := time.Now()  
expensiveCall()  
t1 := time.Now()  
fmt.Printf("The call took %v to run.\n", t1.Sub(t0))
```

func [ParseDuration](#)

```
func ParseDuration(s string) (Duration, error)
```

ParseDuration parses a duration string. A duration string is a possibly signed sequence of decimal numbers, each with optional fraction and a unit suffix, such as "300ms", "-1.5h" or "2h45m". Valid time units are "ns", "us" (or "?s"), "ms", "s", "m", "h".

func [Since](#)

```
func Since(t Time) Duration
```

Since returns the time elapsed since t. It is shorthand for `time.Now().Sub(t)`.

func (Duration) [Hours](#)

```
func (d Duration) Hours() float64
```

Hours returns the duration as a floating point number of hours.

func (Duration) [Minutes](#)

```
func (d Duration) Minutes() float64
```

Minutes returns the duration as a floating point number of minutes.

func (Duration) [Nanoseconds](#)

```
func (d Duration) Nanoseconds() int64
```

Nanoseconds returns the duration as an integer nanosecond count.

func (Duration) [Seconds](#)

```
func (d Duration) Seconds() float64
```

Seconds returns the duration as a floating point number of seconds.

func (Duration) [String](#)

```
func (d Duration) String() string
```

String returns a string representing the duration in the form "72h3m0.5s". Leading zero units are omitted. As a special case, durations less than one second format use a smaller unit (milli-, micro-, or nanoseconds) to ensure that the leading digit is non-zero. The zero duration formats as 0, with no unit.

type [Location](#)

```
type Location struct {  
    // contains filtered or unexported fields  
}
```

A Location maps time instants to the zone in use at that time. Typically, the Location represents the collection of time offsets in use in a geographical area, such as CEST and CET for central Europe.

```
var Local *Location = &localLoc
```

Local represents the system's local time zone.

```
var UTC *Location = &utcLoc
```

UTC represents Universal Coordinated Time (UTC).

func [FixedZone](#)

```
func FixedZone(name string, offset int) *Location
```

FixedZone returns a Location that always uses the given zone name and offset (seconds east of UTC).

func [LoadLocation](#)

```
func LoadLocation(name string) (*Location, error)
```

LoadLocation returns the Location with the given name.

If the name is "" or "UTC", LoadLocation returns UTC. If the name is "Local", LoadLocation returns Local.

Otherwise, the name is taken to be a location name corresponding to a file in the IANA Time Zone database, such as "America/New_York".

The time zone database needed by LoadLocation may not be present on all systems, especially non-Unix systems. LoadLocation looks in the directory or uncompressed zip file named by the ZONEINFO environment variable, if any,

then looks in known installation locations on Unix systems, and finally looks in \$GOROOT/lib/time/zoneinfo.zip.

func (*Location) [String](#)

```
func (l *Location) String() string
```

String returns a descriptive name for the time zone information, corresponding to the argument to LoadLocation.

type [Month](#)

```
type Month int
```

A Month specifies a month of the year (January = 1, ...).

```
const (  
    January Month = 1 + iota  
    February  
    March  
    April  
    May  
    June  
    July  
    August  
    September  
    October  
    November  
    December  
)
```

? Example

? Example

Code:

```
_, month, day := time.Now().Date()  
if month == time.November && day == 10 {  
    fmt.Println("Happy Go day!")  
}
```

func (Month) [String](#)

```
func (m Month) String() string
```

String returns the English name of the month ("January", "February", ...).

type [ParseError](#)

```
type ParseError struct {  
    Layout      string  
    Value       string  
    LayoutElem  string  
    ValueElem   string  
    Message     string  
}
```

ParseError describes a problem parsing a time string.

func (*ParseError) [Error](#)

```
func (e *ParseError) Error() string
```

Error returns the string representation of a ParseError.

type [Ticker](#)

```
type Ticker struct {  
    C <-chan Time // The channel on which the ticks are delivered.  
    // contains filtered or unexported fields  
}
```

A Ticker holds a synchronous channel that delivers `ticks' of a clock at intervals.

func [NewTicker](#)

```
func NewTicker(d Duration) *Ticker
```

NewTicker returns a new Ticker containing a channel that will send the time with a period specified by the duration argument. It adjusts the intervals or drops ticks to make up for slow receivers. The duration d must be greater than zero; if not, NewTicker will panic.

func (***Ticker**) [Stop](#)

```
func (t *Ticker) Stop()
```

Stop turns off a ticker. After Stop, no more ticks will be sent.

type [Time](#)

```
type Time struct {  
    // contains filtered or unexported fields  
}
```

A `Time` represents an instant in time with nanosecond precision.

Programs using times should typically store and pass them as values, not pointers. That is, time variables and struct fields should be of type `time.Time`, not `*time.Time`. A `Time` value can be used by multiple goroutines simultaneously.

Time instants can be compared using the `Before`, `After`, and `Equal` methods. The `Sub` method subtracts two instants, producing a `Duration`. The `Add` method adds a `Time` and a `Duration`, producing a `Time`.

The zero value of type `Time` is January 1, year 1, 00:00:00.000000000 UTC. As this time is unlikely to come up in practice, the `IsZero` method gives a simple way of detecting a time that has not been initialized explicitly.

Each `Time` has associated with it a `Location`, consulted when computing the presentation form of the time, such as in the `Format`, `Hour`, and `Year` methods. The methods `Local`, `UTC`, and `In` return a `Time` with a specific location. Changing the location in this way changes only the presentation; it does not change the instant in time being denoted and therefore does not affect the computations described in earlier paragraphs.

func [Date](#)

```
func Date(year int, month Month, day, hour, min, sec, nsec int, loc
```

`Date` returns the `Time` corresponding to

```
yyyy-mm-dd hh:mm:ss + nsec nanoseconds
```

in the appropriate zone for that time in the given location.

The month, day, hour, min, sec, and nsec values may be outside their usual

ranges and will be normalized during the conversion. For example, October 32 converts to November 1.

A daylight savings time transition skips or repeats times. For example, in the United States, March 13, 2011 2:15am never occurred, while November 6, 2011 1:15am occurred twice. In such cases, the choice of time zone, and therefore the time, is not well-defined. `Date` returns a time that is correct in one of the two zones involved in the transition, but it does not guarantee which.

`Date` panics if `loc` is `nil`.

? Example

? Example

Code:

```
t := time.Date(2009, time.November, 10, 23, 0, 0, 0, time.UTC)
fmt.Printf("Go launched at %s\n", t.Local())
```

Output:

```
Go launched at 2009-11-10 15:00:00 -0800 PST
```

func [Now](#)

```
func Now() Time
```

`Now` returns the current local time.

func [Parse](#)

```
func Parse(layout, value string) (Time, error)
```

`Parse` parses a formatted string and returns the time value it represents. The layout defines the format by showing the representation of the standard time,

```
Mon Jan 2 15:04:05 -0700 MST 2006
```

which is then used to describe the string to be parsed. Predefined layouts `ANSIC`, `UnixDate`, `RFC3339` and others describe standard representations. For

more information about the formats and the definition of the standard time, see the documentation for ANSIC.

Elements omitted from the value are assumed to be zero or, when zero is impossible, one, so parsing "3:04pm" returns the time corresponding to Jan 1, year 0, 15:04:00 UTC. Years must be in the range 0000..9999. The day of the week is checked for syntax but it is otherwise ignored.

func [Unix](#)

```
func Unix(sec int64, nsec int64) Time
```

Unix returns the local Time corresponding to the given Unix time, sec seconds and nsec nanoseconds since January 1, 1970 UTC. It is valid to pass nsec outside the range [0, 999999999].

func (Time) [Add](#)

```
func (t Time) Add(d Duration) Time
```

Add returns the time t+d.

func (Time) [AddDate](#)

```
func (t Time) AddDate(years int, months int, days int) Time
```

AddDate returns the time corresponding to adding the given number of years, months, and days to t. For example, AddDate(-1, 2, 3) applied to January 1, 2011 returns March 4, 2010.

AddDate normalizes its result in the same way that Date does, so, for example, adding one month to October 31 yields December 1, the normalized form for November 31.

func (Time) [After](#)

```
func (t Time) After(u Time) bool
```

After reports whether the time instant t is after u.

func (Time) [Before](#)

```
func (t Time) Before(u Time) bool
```

Before reports whether the time instant `t` is before `u`.

func (Time) [Clock](#)

```
func (t Time) Clock() (hour, min, sec int)
```

Clock returns the hour, minute, and second within the day specified by `t`.

func (Time) [Date](#)

```
func (t Time) Date() (year int, month Month, day int)
```

Date returns the year, month, and day in which `t` occurs.

func (Time) [Day](#)

```
func (t Time) Day() int
```

Day returns the day of the month specified by `t`.

func (Time) [Equal](#)

```
func (t Time) Equal(u Time) bool
```

Equal reports whether `t` and `u` represent the same time instant. Two times can be equal even if they are in different locations. For example, 6:00 +0200 CEST and 4:00 UTC are Equal. This comparison is different from using `t == u`, which also compares the locations.

func (Time) [Format](#)

```
func (t Time) Format(layout string) string
```

Format returns a textual representation of the time value formatted according to `layout`. The layout defines the format by showing the representation of the standard time,

Mon Jan 2 15:04:05 -0700 MST 2006

which is then used to describe the time to be formatted. Predefined layouts ANSIC, UnixDate, RFC3339 and others describe standard representations. For more information about the formats and the definition of the standard time, see the documentation for ANSIC.

func (*Time) [GobDecode](#)

```
func (t *Time) GobDecode(buf []byte) error
```

GobDecode implements the gob.GobDecoder interface.

func (Time) [GobEncode](#)

```
func (t Time) GobEncode() ([]byte, error)
```

GobEncode implements the gob.GobEncoder interface.

func (Time) [Hour](#)

```
func (t Time) Hour() int
```

Hour returns the hour within the day specified by t, in the range [0, 23].

func (Time) [ISOWeek](#)

```
func (t Time) ISOWeek() (year, week int)
```

ISOWeek returns the ISO 8601 year and week number in which t occurs. Week ranges from 1 to 53. Jan 01 to Jan 03 of year n might belong to week 52 or 53 of year n-1, and Dec 29 to Dec 31 might belong to week 1 of year n+1.

func (Time) [In](#)

```
func (t Time) In(loc *Location) Time
```

In returns t with the location information set to loc.

In panics if loc is nil.

func (Time) [IsZero](#)

```
func (t Time) IsZero() bool
```

IsZero reports whether t represents the zero time instant, January 1, year 1, 00:00:00 UTC.

func (Time) [Local](#)

```
func (t Time) Local() Time
```

Local returns t with the location set to local time.

func (Time) [Location](#)

```
func (t Time) Location() *Location
```

Location returns the time zone information associated with t.

func (Time) [MarshalJSON](#)

```
func (t Time) MarshalJSON() ([]byte, error)
```

MarshalJSON implements the json.Marshaler interface. Time is formatted as RFC3339.

func (Time) [Minute](#)

```
func (t Time) Minute() int
```

Minute returns the minute offset within the hour specified by t, in the range [0, 59].

func (Time) [Month](#)

```
func (t Time) Month() Month
```

Month returns the month of the year specified by t.

func (Time) [Nanosecond](#)

```
func (t Time) Nanosecond() int
```

Nanosecond returns the nanosecond offset within the second specified by t, in the range [0, 999999999].

func (Time) [Second](#)

```
func (t Time) Second() int
```

Second returns the second offset within the minute specified by t, in the range [0, 59].

func (Time) [String](#)

```
func (t Time) String() string
```

String returns the time formatted using the format string

```
"2006-01-02 15:04:05.999999999 -0700 MST"
```

func (Time) [Sub](#)

```
func (t Time) Sub(u Time) Duration
```

Sub returns the duration t-u. To compute t-d for a duration d, use t.Add(-d).

func (Time) [UTC](#)

```
func (t Time) UTC() Time
```

UTC returns t with the location set to UTC.

func (Time) [Unix](#)

```
func (t Time) Unix() int64
```

Unix returns t as a Unix time, the number of seconds elapsed since January 1, 1970 UTC.

func (Time) [UnixNano](#)

```
func (t Time) UnixNano() int64
```

UnixNano returns t as a Unix time, the number of nanoseconds elapsed since January 1, 1970 UTC. The result is undefined if the Unix time in nanoseconds cannot be represented by an int64. Note that this means the result of calling UnixNano on the zero Time is undefined.

func (*Time) [UnmarshalJSON](#)

```
func (t *Time) UnmarshalJSON(data []byte) (err error)
```

UnmarshalJSON implements the json.Unmarshaler interface. Time is expected in RFC3339 format.

func (Time) [Weekday](#)

```
func (t Time) Weekday() Weekday
```

Weekday returns the day of the week specified by t.

func (Time) [Year](#)

```
func (t Time) Year() int
```

Year returns the year in which t occurs.

func (Time) [Zone](#)

```
func (t Time) Zone() (name string, offset int)
```

Zone computes the time zone in effect at time t, returning the abbreviated name of the zone (such as "CET") and its offset in seconds east of UTC.

type [Timer](#)

```
type Timer struct {  
    C <-chan Time  
    // contains filtered or unexported fields  
}
```

The `Timer` type represents a single event. When the `Timer` expires, the current time will be sent on `C`, unless the `Timer` was created by `AfterFunc`.

func [AfterFunc](#)

```
func AfterFunc(d Duration, f func()) *Timer
```

`AfterFunc` waits for the duration to elapse and then calls `f` in its own goroutine. It returns a `Timer` that can be used to cancel the call using its `Stop` method.

func [NewTimer](#)

```
func NewTimer(d Duration) *Timer
```

`NewTimer` creates a new `Timer` that will send the current time on its channel after at least duration `d`.

func (*Timer) [Stop](#)

```
func (t *Timer) Stop() (ok bool)
```

`Stop` prevents the `Timer` from firing. It returns `true` if the call stops the timer, `false` if the timer has already expired or stopped.

type [Weekday](#)

```
type Weekday int
```

A Weekday specifies a day of the week (Sunday = 0, ...).

```
const (  
    Sunday Weekday = iota  
    Monday  
    Tuesday  
    Wednesday  
    Thursday  
    Friday  
    Saturday  
)
```

func (Weekday) [String](#)

```
func (d Weekday) String() string
```

String returns the English name of the day ("Sunday", "Monday", ...).

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package unicode

```
import "unicode"
```

[Overview](#)

[Index](#)

[Subdirectories](#)

Overview ?

Overview ?

Package unicode provides data and functions to test some properties of Unicode code points.

Index

Constants

Variables

[func Is\(rangeTab *RangeTable, r rune\) bool](#)

[func IsControl\(r rune\) bool](#)

[func IsDigit\(r rune\) bool](#)

[func IsGraphic\(r rune\) bool](#)

[func IsLetter\(r rune\) bool](#)

[func IsLower\(r rune\) bool](#)

[func IsMark\(r rune\) bool](#)

[func IsNumber\(r rune\) bool](#)

[func IsOneOf\(set \[\]*RangeTable, r rune\) bool](#)

[func IsPrint\(r rune\) bool](#)

[func IsPunct\(r rune\) bool](#)

[func IsSpace\(r rune\) bool](#)

[func IsSymbol\(r rune\) bool](#)

[func IsTitle\(r rune\) bool](#)

[func IsUpper\(r rune\) bool](#)

[func SimpleFold\(r rune\) rune](#)

[func To\(_ case int, r rune\) rune](#)

[func ToLower\(r rune\) rune](#)

[func ToTitle\(r rune\) rune](#)

[func ToUpper\(r rune\) rune](#)

[type CaseRange](#)

[type Range16](#)

[type Range32](#)

[type RangeTable](#)

[type SpecialCase](#)

[func \(special SpecialCase\) ToLower\(r rune\) rune](#)

[func \(special SpecialCase\) ToTitle\(r rune\) rune](#)

[func \(special SpecialCase\) ToUpper\(r rune\) rune](#)

Bugs

Package files

[casetables.go](#) [digit.go](#) [graphic.go](#) [letter.go](#) [tables.go](#)

Constants

```
const (  
    MaxRune          = '\U0010FFFF' // Maximum valid Unicode code poi  
    ReplacementChar = '\uFFFD'     // Represents invalid code points  
    MaxASCII        = '\u007F'     // maximum ASCII value.  
    MaxLatin1       = '\u00FF'     // maximum Latin-1 value.  
)
```

```
const (  
    UpperCase = iota  
    LowerCase  
    TitleCase  
    MaxCase  
)
```

Indices into the Delta arrays inside CaseRanges for case mapping.

```
const (  
    UpperLower = MaxRune + 1 // (Cannot be a valid delta.)  
)
```

If the Delta field of a CaseRange is UpperLower or LowerUpper, it means this CaseRange represents a sequence of the form (say) Upper Lower Upper Lower.

```
const Version = "6.0.0"
```

Version is the Unicode edition from which the tables are derived.

Variables

```
var (  
  Cc      = _Cc // Cc is the set of Unicode characters in category  
  Cf      = _Cf // Cf is the set of Unicode characters in category  
  Co      = _Co // Co is the set of Unicode characters in category  
  Cs      = _Cs // Cs is the set of Unicode characters in category  
  Digit   = _Nd // Digit is the set of Unicode characters with the  
  Nd      = _Nd // Nd is the set of Unicode characters in category  
  Letter  = _L  // Letter/L is the set of Unicode letters, category  
  L       = _L  
  Lm      = _Lm // Lm is the set of Unicode characters in category  
  Lo      = _Lo // Lo is the set of Unicode characters in category  
  Lower   = _Ll // Lower is the set of Unicode lower case letters.  
  Ll      = _Ll // Ll is the set of Unicode characters in category  
  Mark    = _M  // Mark/M is the set of Unicode mark characters, ca  
  M       = _M  
  Mc      = _Mc // Mc is the set of Unicode characters in category  
  Me      = _Me // Me is the set of Unicode characters in category  
  Mn      = _Mn // Mn is the set of Unicode characters in category  
  Nl      = _Nl // Nl is the set of Unicode characters in category  
  No      = _No // No is the set of Unicode characters in category  
  Number  = _N  // Number/N is the set of Unicode number characters  
  N       = _N  
  Other   = _C  // Other/C is the set of Unicode control and special  
  C       = _C  
  Pc      = _Pc // Pc is the set of Unicode characters in category  
  Pd      = _Pd // Pd is the set of Unicode characters in category  
  Pe      = _Pe // Pe is the set of Unicode characters in category  
  Pf      = _Pf // Pf is the set of Unicode characters in category  
  Pi      = _Pi // Pi is the set of Unicode characters in category  
  Po      = _Po // Po is the set of Unicode characters in category  
  Ps      = _Ps // Ps is the set of Unicode characters in category  
  Punct   = _P  // Punct/P is the set of Unicode punctuation charac  
  P       = _P  
  Sc      = _Sc // Sc is the set of Unicode characters in category  
  Sk      = _Sk // Sk is the set of Unicode characters in category  
  Sm      = _Sm // Sm is the set of Unicode characters in category  
  So      = _So // So is the set of Unicode characters in category  
  Space   = _Z  // Space/Z is the set of Unicode space characters,  
  Z       = _Z  
  Symbol  = _S  // Symbol/S is the set of Unicode symbol characters,  
  S       = _S  
  Title   = _Lt // Title is the set of Unicode title case letters.  
  Lt      = _Lt // Lt is the set of Unicode characters in category  
  Upper   = _Lu // Upper is the set of Unicode upper case letters.  
  Lu      = _Lu // Lu is the set of Unicode characters in category
```

```

Zl      = _Zl // Zl is the set of Unicode characters in category
Zp      = _Zp // Zp is the set of Unicode characters in category
Zs      = _Zs // Zs is the set of Unicode characters in category
)

```

The following variables are of type *RangeTable:

```

var (
  Arabic           = _Arabic           // Arabic is th
  Armenian         = _Armenian         // Armenian is
  Avestan          = _Avestan          // Avestan is t
  Balinese         = _Balinese         // Balinese is
  Bamum            = _Bamum            // Bamum is the
  Batak            = _Batak            // Batak is the
  Bengali          = _Bengali          // Bengali is t
  Bopomofo         = _Bopomofo         // Bopomofo is
  Brahmi           = _Brahmi           // Brahmi is th
  Braille          = _Braille          // Braille is t
  Buginese         = _Buginese         // Buginese is
  Buhid            = _Buhid            // Buhid is the
  Canadian_Aboriginal = _Canadian_Aboriginal // Canadian_Abo
  Carian           = _Carian           // Carian is th
  Cham             = _Cham             // Cham is the
  Cherokee         = _Cherokee         // Cherokee is
  Common           = _Common           // Common is th
  Coptic           = _Coptic           // Coptic is th
  Cuneiform        = _Cuneiform        // Cuneiform is
  Cypriot          = _Cypriot          // Cypriot is t
  Cyrillic         = _Cyrillic         // Cyrillic is
  Deseret          = _Deseret          // Deseret is t
  Devanagari       = _Devanagari       // Devanagari i
  Egyptian_Hieroglyphs = _Egyptian_Hieroglyphs // Egyptian_Hie
  Ethiopic         = _Ethiopic         // Ethiopic is
  Georgian         = _Georgian         // Georgian is
  Glagolitic       = _Glagolitic       // Glagolitic i
  Gothic           = _Gothic           // Gothic is th
  Greek            = _Greek            // Greek is the
  Gujarati         = _Gujarati         // Gujarati is
  Gurmukhi         = _Gurmukhi         // Gurmukhi is
  Han              = _Han              // Han is the s
  Hangul           = _Hangul           // Hangul is th
  Hanunoo          = _Hanunoo          // Hanunoo is t
  Hebrew           = _Hebrew           // Hebrew is th
  Hiragana         = _Hiragana         // Hiragana is
  Imperial_Aramaic = _Imperial_Aramaic // Imperial_Ara
  Inherited        = _Inherited        // Inherited is
  Inscriptional_Pahlavi = _Inscriptional_Pahlavi // Inscriptio
  Inscriptional_Parthian = _Inscriptional_Parthian // Inscriptio
  Javanese         = _Javanese         // Javanese is

```

Kaithi	= _Kaithi	// Kaithi is th
Kannada	= _Kannada	// Kannada is t
Katakana	= _Katakana	// Katakana is
Kayah_Li	= _Kayah_Li	// Kayah_Li is
Kharoshthi	= _Kharoshthi	// Kharoshthi i
Khmer	= _Khmer	// Khmer is the
Lao	= _Lao	// Lao is the s
Latin	= _Latin	// Latin is the
Lepcha	= _Lepcha	// Lepcha is th
Limbu	= _Limbu	// Limbu is the
Linear_B	= _Linear_B	// Linear_B is
Lisu	= _Lisu	// Lisu is the
Lycian	= _Lycian	// Lycian is th
Lydian	= _Lydian	// Lydian is th
Malayalam	= _Malayalam	// Malayalam is
Mandaic	= _Mandaic	// Mandaic is t
Meetei_Mayek	= _Meetei_Mayek	// Meetei_Mayek
Mongolian	= _Mongolian	// Mongolian is
Myanmar	= _Myanmar	// Myanmar is t
New_Tai_Lue	= _New_Tai_Lue	// New_Tai_Lue
Nko	= _Nko	// Nko is the s
Ogham	= _Ogham	// Ogham is the
Ol_Chiki	= _Ol_Chiki	// Ol_Chiki is
Old_Italic	= _Old_Italic	// Old_Italic i
Old_Persian	= _Old_Persian	// Old_Persian
Old_South_Arabian	= _Old_South_Arabian	// Old_South_Ar
Old_Turkic	= _Old_Turkic	// Old_Turkic i
Oriya	= _Oriya	// Oriya is the
Osmanya	= _Osmanya	// Osmanya is t
Phags_Pa	= _Phags_Pa	// Phags_Pa is
Phoenician	= _Phoenician	// Phoenician i
Rejang	= _Rejang	// Rejang is th
Runic	= _Runic	// Runic is the
Samaritan	= _Samaritan	// Samaritan is
Saurashtra	= _Saurashtra	// Saurashtra i
Shavian	= _Shavian	// Shavian is t
Sinhala	= _Sinhala	// Sinhala is t
Sundanese	= _Sundanese	// Sundanese is
Syloti_Nagri	= _Syloti_Nagri	// Syloti_Nagri
Syriac	= _Syriac	// Syriac is th
Tagalog	= _Tagalog	// Tagalog is t
Tagbanwa	= _Tagbanwa	// Tagbanwa is
Tai_Le	= _Tai_Le	// Tai_Le is th
Tai_Tham	= _Tai_Tham	// Tai_Tham is
Tai_Viet	= _Tai_Viet	// Tai_Viet is
Tamil	= _Tamil	// Tamil is the
Telugu	= _Telugu	// Telugu is th
Thaana	= _Thaana	// Thaana is th
Thai	= _Thai	// Thai is the
Tibetan	= _Tibetan	// Tibetan is t

```

    Tifinagh          = _Tifinagh          // Tifinagh is
    Ugaritic          = _Ugaritic          // Ugaritic is
    Vai               = _Vai              // Vai is the s
    Yi                = _Yi              // Yi is the se
)

```

The following variables are of type *RangeTable:

```

var (
    ASCII_Hex_Digit      = _ASCII_Hex_Digit
    Bidi_Control         = _Bidi_Control
    Dash                 = _Dash
    Deprecated           = _Deprecated
    Diacritic            = _Diacritic
    Extender             = _Extender
    Hex_Digit           = _Hex_Digit
    Hyphen               = _Hyphen
    IDS_Binary_Operator  = _IDS_Binary_Operator
    IDS_Tertiary_Operator = _IDS_Tertiary_Operator
    Ideographic          = _Ideographic
    Join_Control         = _Join_Control
    Logical_Order_Exception = _Logical_Order_Exception
    Noncharacter_Code_Point = _Noncharacter_Code_Point
    Other_Alphabetic     = _Other_Alphabetic
    Other_Default_Ignorable_Code_Point = _Other_Default_Ignorable_Co
    Other_Grapheme_Extend = _Other_Grapheme_Extend
    Other_ID_Continue    = _Other_ID_Continue
    Other_ID_Start       = _Other_ID_Start
    Other_Lowercase      = _Other_Lowercase
    Other_Math           = _Other_Math
    Other_Uppercase      = _Other_Uppercase
    Pattern_Syntax       = _Pattern_Syntax
    Pattern_White_Space  = _Pattern_White_Space
    Quotation_Mark       = _Quotation_Mark
    Radical              = _Radical
    STerm                = _STerm
    Soft_Dotted          = _Soft_Dotted
    Terminal_Punctuation = _Terminal_Punctuation
    Unified_Ideograph    = _Unified_Ideograph
    Variation_Selector   = _Variation_Selector
    White_Space          = _White_Space
)

```

The following variables are of type *RangeTable:

```
var CaseRanges = _CaseRanges
```

CaseRanges is the table describing case mappings for all letters with non-self

mappings.

```
var Categories = map[string]*RangeTable{
    "C": C,
    "Cc": Cc,
    "Cf": Cf,
    "Co": Co,
    "Cs": Cs,
    "L": L,
    "Ll": Ll,
    "Lm": Lm,
    "Lo": Lo,
    "Lt": Lt,
    "Lu": Lu,
    "M": M,
    "Mc": Mc,
    "Me": Me,
    "Mn": Mn,
    "N": N,
    "Nd": Nd,
    "Nl": Nl,
    "No": No,
    "P": P,
    "Pc": Pc,
    "Pd": Pd,
    "Pe": Pe,
    "Pf": Pf,
    "Pi": Pi,
    "Po": Po,
    "Ps": Ps,
    "S": S,
    "Sc": Sc,
    "Sk": Sk,
    "Sm": Sm,
    "So": So,
    "Z": Z,
    "Zl": Zl,
    "Zp": Zp,
    "Zs": Zs,
}
```

Categories is the set of Unicode category tables.

```
var FoldCategory = map[string]*RangeTable{
    "Common": foldCommon,
    "Greek": foldGreek,
    "Inherited": foldInherited,
    "L": foldL,
    "Ll": foldLl,
```

```

    "Lt":      foldLt,
    "Lu":      foldLu,
    "M":       foldM,
    "Mn":      foldMn,
}

```

FoldCategory maps a category name to a table of code points outside the category that are equivalent under simple case folding to code points inside the category. If there is no entry for a category name, there are no such points.

```
var FoldScript = map[string]*RangeTable{}
```

FoldScript maps a script name to a table of code points outside the script that are equivalent under simple case folding to code points inside the script. If there is no entry for a script name, there are no such points.

```
var GraphicRanges = []*RangeTable{
    L, M, N, P, S, Zs,
}

```

GraphicRanges defines the set of graphic characters according to Unicode.

```
var PrintRanges = []*RangeTable{
    L, M, N, P, S,
}

```

PrintRanges defines the set of printable characters according to Go. ASCII space, U+0020, is handled separately.

```
var Properties = map[string]*RangeTable{
    "ASCII_Hex_Digit":    ASCII_Hex_Digit,
    "Bidi_Control":      Bidi_Control,
    "Dash":              Dash,
    "Deprecated":        Deprecated,
    "Diacritic":         Diacritic,
    "Extender":          Extender,
    "Hex_Digit":         Hex_Digit,
    "Hyphen":            Hyphen,
    "IDS_Binary_Operator": IDS_Binary_Operator,
    "IDS_Tertiary_Operator": IDS_Tertiary_Operator,
    "Ideographic":      Ideographic,
    "Join_Control":     Join_Control,
    "Logical_Order_Exception": Logical_Order_Exception,
    "Noncharacter_Code_Point": Noncharacter_Code_Point,
    "Other_Alphabetic":  Other_Alphabetic,
    "Other_Default_Ignorable_Code_Point": Other_Default_Ignorable_Co
}

```

```

"Other_Grapheme_Extend":      Other_Grapheme_Extend,
"Other_ID_Continue":          Other_ID_Continue,
"Other_ID_Start":             Other_ID_Start,
"Other_Lowercase":           Other_Lowercase,
"Other_Math":                 Other_Math,
"Other_Uppercase":           Other_Uppercase,
"Pattern_Syntax":            Pattern_Syntax,
"Pattern_White_Space":       Pattern_White_Space,
"Quotation_Mark":            Quotation_Mark,
"Radical":                   Radical,
"STerm":                     STerm,
"Soft_Dotted":               Soft_Dotted,
"Terminal_Punctuation":      Terminal_Punctuation,
"Unified_Ideograph":         Unified_Ideograph,
"Variation_Selector":        Variation_Selector,
"White_Space":               White_Space,
}

```

Properties is the set of Unicode property tables.

```

var Scripts = map[string]*RangeTable{
  "Arabic":           Arabic,
  "Armenian":         Armenian,
  "Avestan":          Avestan,
  "Balinese":         Balinese,
  "Bamum":            Bamum,
  "Batak":            Batak,
  "Bengali":          Bengali,
  "Bopomofo":         Bopomofo,
  "Brahmi":           Brahmi,
  "Braille":          Braille,
  "Buginese":         Buginese,
  "Buhid":            Buhid,
  "Canadian_Aboriginal": Canadian_Aboriginal,
  "Carian":           Carian,
  "Cham":             Cham,
  "Cherokee":         Cherokee,
  "Common":           Common,
  "Coptic":           Coptic,
  "Cuneiform":        Cuneiform,
  "Cypriot":          Cypriot,
  "Cyrillic":         Cyrillic,
  "Deseret":          Deseret,
  "Devanagari":       Devanagari,
  "Egyptian_Hieroglyphs": Egyptian_Hieroglyphs,
  "Ethiopic":         Ethiopic,
  "Georgian":         Georgian,
  "Glagolitic":       Glagolitic,
  "Gothic":           Gothic,
}

```

"Greek":	Greek,
"Gujarati":	Gujarati,
"Gurmukhi":	Gurmukhi,
"Han":	Han,
"Hangul":	Hangul,
"Hanunoo":	Hanunoo,
"Hebrew":	Hebrew,
"Hiragana":	Hiragana,
"Imperial_Aramaic":	Imperial_Aramaic,
"Inherited":	Inherited,
"Inscriptional_Pahlavi":	Inscriptional_Pahlavi,
"Inscriptional_Parthian":	Inscriptional_Parthian,
"Javanese":	Javanese,
"Kaithi":	Kaithi,
"Kannada":	Kannada,
"Katakana":	Katakana,
"Kayah_Li":	Kayah_Li,
"Kharoshthi":	Kharoshthi,
"Khmer":	Khmer,
"Lao":	Lao,
"Latin":	Latin,
"Lepcha":	Lepcha,
"Limbu":	Limbu,
"Linear_B":	Linear_B,
"Lisu":	Lisu,
"Lycian":	Lycian,
"Lydian":	Lydian,
"Malayalam":	Malayalam,
"Mandaic":	Mandaic,
"Meetei_Mayek":	Meetei_Mayek,
"Mongolian":	Mongolian,
"Myanmar":	Myanmar,
"New_Tai_Lue":	New_Tai_Lue,
"Nko":	Nko,
"Ogham":	Ogham,
"Ol_Chiki":	Ol_Chiki,
"Old_Italic":	Old_Italic,
"Old_Persian":	Old_Persian,
"Old_South_Arabian":	Old_South_Arabian,
"Old_Turkic":	Old_Turkic,
"Oriya":	Oriya,
"Osmanya":	Osmanya,
"Phags_Pa":	Phags_Pa,
"Phoenician":	Phoenician,
"Rejang":	Rejang,
"Runic":	Runic,
"Samaritan":	Samaritan,
"Saurashtra":	Saurashtra,
"Shavian":	Shavian,
"Sinhala":	Sinhala,

```
"Sundanese": Sundanese,  
"Syloti_Nagri": Syloti_Nagri,  
"Syriac": Syriac,  
"Tagalog": Tagalog,  
"Tagbanwa": Tagbanwa,  
"Tai_Le": Tai_Le,  
"Tai_Tham": Tai_Tham,  
"Tai_Viet": Tai_Viet,  
"Tamil": Tamil,  
"Telugu": Telugu,  
"Thaana": Thaana,  
"Thai": Thai,  
"Tibetan": Tibetan,  
"Tifinagh": Tifinagh,  
"Ugaritic": Ugaritic,  
"Vai": Vai,  
"Yi": Yi,  
}
```

Scripts is the set of Unicode script tables.

func [Is](#)

```
func Is(rangeTab *RangeTable, r rune) bool
```

Is tests whether rune is in the specified table of ranges.

func [IsControl](#)

```
func IsControl(r rune) bool
```

IsControl reports whether the rune is a control character. The C (Other) Unicode category includes more code points such as surrogates; use Is(C, r) to test for them.

func [IsDigit](#)

```
func IsDigit(r rune) bool
```

IsDigit reports whether the rune is a decimal digit.

func [IsGraphic](#)

```
func IsGraphic(r rune) bool
```

IsGraphic reports whether the rune is defined as a Graphic by Unicode. Such characters include letters, marks, numbers, punctuation, symbols, and spaces, from categories L, M, N, P, S, Zs.

func IsLetter

```
func IsLetter(r rune) bool
```

IsLetter reports whether the rune is a letter (category L).

func IsLower

```
func IsLower(r rune) bool
```

IsLower reports whether the rune is a lower case letter.

func IsMark

```
func IsMark(r rune) bool
```

IsMark reports whether the rune is a mark character (category M).

func IsNumber

```
func IsNumber(r rune) bool
```

IsNumber reports whether the rune is a number (category N).

func [IsOneOf](#)

```
func IsOneOf(set []*RangeTable, r rune) bool
```

IsOneOf reports whether the rune is a member of one of the ranges.

func [IsPrint](#)

```
func IsPrint(r rune) bool
```

IsPrint reports whether the rune is defined as printable by Go. Such characters include letters, marks, numbers, punctuation, symbols, and the ASCII space character, from categories L, M, N, P, S and the ASCII space character. This categorization is the same as IsGraphic except that the only spacing character is ASCII space, U+0020.

func [IsPunct](#)

```
func IsPunct(r rune) bool
```

IsPunct reports whether the rune is a Unicode punctuation character (category P).

func IsSpace

```
func IsSpace(r rune) bool
```

IsSpace reports whether the rune is a space character as defined by Unicode's White Space property; in the Latin-1 space this is

```
'\t', '\n', '\v', '\f', '\r', ' ', U+0085 (NEL), U+00A0 (NBSP).
```

Other definitions of spacing characters are set by category Z and property Pattern_White_Space.

func IsSymbol

```
func IsSymbol(r rune) bool
```

IsSymbol reports whether the rune is a symbolic character.

func [IsTitle](#)

```
func IsTitle(r rune) bool
```

IsTitle reports whether the rune is a title case letter.

func [IsUpper](#)

```
func IsUpper(r rune) bool
```

IsUpper reports whether the rune is an upper case letter.

func [SimpleFold](#)

```
func SimpleFold(r rune) rune
```

SimpleFold iterates over Unicode code points equivalent under the Unicode-defined simple case folding. Among the code points equivalent to rune (including rune itself), SimpleFold returns the smallest rune $\geq r$ if one exists, or else the smallest rune ≥ 0 .

For example:

```
SimpleFold('A') = 'a'
```

```
SimpleFold('a') = 'A'
```

```
SimpleFold('K') = 'k'
```

```
SimpleFold('k') = '\u212A' (Kelvin symbol, ?)
```

```
SimpleFold('\u212A') = 'K'
```

```
SimpleFold('1') = '1'
```

func To

```
func To(_case int, r rune) rune
```

To maps the rune to the specified case: UpperCase, LowerCase, or TitleCase.

func ToLower

```
func ToLower(r rune) rune
```

ToLower maps the rune to lower case.

func [ToTitle](#)

```
func ToTitle(r rune) rune
```

ToTitle maps the rune to title case.

func ToUpper

```
func ToUpper(r rune) rune
```

ToUpper maps the rune to upper case.

type [CaseRange](#)

```
type CaseRange struct {  
    Lo    uint32  
    Hi    uint32  
    Delta d  
}
```

CaseRange represents a range of Unicode code points for simple (one code point to one code point) case conversion. The range runs from Lo to Hi inclusive, with a fixed stride of 1. Deltas are the number to add to the code point to reach the code point for a different case for that character. They may be negative. If zero, it means the character is in the corresponding case. There is a special case representing sequences of alternating corresponding Upper and Lower pairs. It appears with a fixed Delta of

```
{UpperLower, UpperLower, UpperLower}
```

The constant UpperLower has an otherwise impossible delta value.

type [Range16](#)

```
type Range16 struct {  
    Lo      uint16  
    Hi      uint16  
    Stride  uint16  
}
```

Range16 represents of a range of 16-bit Unicode code points. The range runs from Lo to Hi inclusive and has the specified stride.

type [Range32](#)

```
type Range32 struct {  
    Lo      uint32  
    Hi      uint32  
    Stride  uint32  
}
```

Range32 represents a range of Unicode code points and is used when one or more of the values will not fit in 16 bits. The range runs from Lo to Hi inclusive and has the specified stride. Lo and Hi must always be $\geq 1 \ll 16$.

type [RangeTable](#)

```
type RangeTable struct {  
    R16 []Range16  
    R32 []Range32  
}
```

RangeTable defines a set of Unicode code points by listing the ranges of code points within the set. The ranges are listed in two slices to save space: a slice of 16-bit ranges and a slice of 32-bit ranges. The two slices must be in sorted order and non-overlapping. Also, R32 should contain only values $\geq 0x10000$ ($1 \ll 16$).

type [SpecialCase](#)

```
type SpecialCase []CaseRange
```

SpecialCase represents language-specific case mappings such as Turkish. Methods of SpecialCase customize (by overriding) the standard mappings.

```
var AzeriCase SpecialCase = _TurkishCase  
var TurkishCase SpecialCase = _TurkishCase
```

func (SpecialCase) [ToLower](#)

```
func (special SpecialCase) ToLower(r rune) rune
```

ToLower maps the rune to lower case giving priority to the special mapping.

func (SpecialCase) [ToTitle](#)

```
func (special SpecialCase) ToTitle(r rune) rune
```

ToTitle maps the rune to title case giving priority to the special mapping.

func (SpecialCase) [ToUpper](#)

```
func (special SpecialCase) ToUpper(r rune) rune
```

ToUpper maps the rune to upper case giving priority to the special mapping.

Bugs

There is no mechanism for full case folding, that is, for characters that involve multiple runes in the input or output.

Subdirectories

Name **Synopsis**

[utf16](#) Package utf16 implements encoding and decoding of UTF-16 sequences.

[utf8](#) Package utf8 implements functions and constants to support text encoded in UTF-8.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Package utf16

```
import "unicode/utf16"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `utf16` implements encoding and decoding of UTF-16 sequences.

Index

[func Decode\(s \[\]uint16\) \[\]rune](#)
[func DecodeRune\(r1, r2 rune\) rune](#)
[func Encode\(s \[\]rune\) \[\]uint16](#)
[func EncodeRune\(r rune\) \(r1, r2 rune\)](#)
[func IsSurrogate\(r rune\) bool](#)

Package files

[utf16.go](#)

func [Decode](#)

```
func Decode(s []uint16) []rune
```

Decode returns the Unicode code point sequence represented by the UTF-16 encoding s.

func [DecodeRune](#)

```
func DecodeRune(r1, r2 rune) rune
```

DecodeRune returns the UTF-16 decoding of a surrogate pair. If the pair is not a valid UTF-16 surrogate pair, DecodeRune returns the Unicode replacement code point U+FFFD.

func [Encode](#)

```
func Encode(s []rune) []uint16
```

Encode returns the UTF-16 encoding of the Unicode code point sequence s.

func [EncodeRune](#)

```
func EncodeRune(r rune) (r1, r2 rune)
```

EncodeRune returns the UTF-16 surrogate pair r1, r2 for the given rune. If the rune is not a valid Unicode code point or does not need encoding, EncodeRune returns U+FFFD, U+FFFD.

func [IsSurrogate](#)

```
func IsSurrogate(r rune) bool
```

IsSurrogate returns true if the specified Unicode code point can appear in a surrogate pair.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package utf8

```
import "unicode/utf8"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package `utf8` implements functions and constants to support text encoded in UTF-8. It includes functions to translate between runes and UTF-8 byte sequences.

Index

Constants

[func DecodeLastRune\(p \[\]byte\) \(r rune, size int\)](#)

[func DecodeLastRuneInString\(s string\) \(r rune, size int\)](#)

[func DecodeRune\(p \[\]byte\) \(r rune, size int\)](#)

[func DecodeRuneInString\(s string\) \(r rune, size int\)](#)

[func EncodeRune\(p \[\]byte, r rune\) int](#)

[func FullRune\(p \[\]byte\) bool](#)

[func FullRuneInString\(s string\) bool](#)

[func RuneCount\(p \[\]byte\) int](#)

[func RuneCountInString\(s string\) \(n int\)](#)

[func RuneLen\(r rune\) int](#)

[func RuneStart\(b byte\) bool](#)

[func Valid\(p \[\]byte\) bool](#)

[func ValidString\(s string\) bool](#)

Package files

[utf8.go](#)

Constants

```
const (  
    RuneError = '\uFFFD'      // the "error" Rune or "Unicode replace  
    RuneSelf  = 0x80         // characters below Runeself are repres  
    MaxRune   = '\U0010FFFF' // Maximum valid Unicode code point.  
    UTFMax    = 4           // maximum number of bytes of a UTF-8 e  
)
```

Numbers fundamental to the encoding.

func DecodeLastRune

```
func DecodeLastRune(p []byte) (r rune, size int)
```

DecodeLastRune unpacks the last UTF-8 encoding in p and returns the rune and its width in bytes. If the encoding is invalid, it returns (RuneError, 1), an impossible result for correct UTF-8.

func [DecodeLastRuneInString](#)

```
func DecodeLastRuneInString(s string) (r rune, size int)
```

DecodeLastRuneInString is like DecodeLastRune but its input is a string. If the encoding is invalid, it returns (RuneError, 1), an impossible result for correct UTF-8.

func [DecodeRune](#)

```
func DecodeRune(p []byte) (r rune, size int)
```

DecodeRune unpacks the first UTF-8 encoding in p and returns the rune and its width in bytes. If the encoding is invalid, it returns (RuneError, 1), an impossible result for correct UTF-8.

func [DecodeRuneInString](#)

```
func DecodeRuneInString(s string) (r rune, size int)
```

DecodeRuneInString is like DecodeRune but its input is a string. If the encoding is invalid, it returns (RuneError, 1), an impossible result for correct UTF-8.

func [EncodeRune](#)

```
func EncodeRune(p []byte, r rune) int
```

EncodeRune writes into p (which must be large enough) the UTF-8 encoding of the rune. It returns the number of bytes written.

func [FullRune](#)

```
func FullRune(p []byte) bool
```

FullRune reports whether the bytes in p begin with a full UTF-8 encoding of a rune. An invalid encoding is considered a full Rune since it will convert as a width-1 error rune.

func [FullRuneInString](#)

```
func FullRuneInString(s string) bool
```

FullRuneInString is like FullRune but its input is a string.

func [RuneCount](#)

```
func RuneCount(p []byte) int
```

RuneCount returns the number of runes in p. Erroneous and short encodings are treated as single runes of width 1 byte.

func [RuneCountInString](#)

```
func RuneCountInString(s string) (n int)
```

RuneCountInString is like RuneCount but its input is a string.

func [RuneLen](#)

```
func RuneLen(r rune) int
```

RuneLen returns the number of bytes required to encode the rune.

func [RuneStart](#)

```
func RuneStart(b byte) bool
```

RuneStart reports whether the byte could be the first byte of an encoded rune. Second and subsequent bytes always have the top two bits set to 10.

func [Valid](#)

```
func Valid(p []byte) bool
```

Valid reports whether p consists entirely of valid UTF-8-encoded runes.

func [ValidString](#)

```
func ValidString(s string) bool
```

ValidString reports whether s consists entirely of valid UTF-8-encoded runes.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Package unsafe

```
import "unsafe"
```

[Overview](#)

[Index](#)

Overview ?

Overview ?

Package unsafe contains operations that step around the type safety of Go programs.

Index

[func Alignof\(v ArbitraryType\) uintptr](#)
[func Offsetof\(v ArbitraryType\) uintptr](#)
[func Sizeof\(v ArbitraryType\) uintptr](#)
[type ArbitraryType](#)
[type Pointer](#)

Package files

[unsafe.go](#)

func [Alignof](#)

```
func Alignof(v ArbitraryType) uintptr
```

Alignof returns the alignment of the value v. It is the maximum value m such that the address of a variable with the type of v will always be zero mod m. If v is of the form structValue.field, it returns the alignment of field f within struct object obj.

func [Offsetof](#)

```
func Offsetof(v ArbitraryType) uintptr
```

Offsetof returns the offset within the struct of the field represented by v, which must be of the form structValue.field. In other words, it returns the number of bytes between the start of the struct and the start of the field.

func [Sizeof](#)

```
func Sizeof(v ArbitraryType) uintptr
```

Sizeof returns the size in bytes occupied by the value v. The size is that of the "top level" of the value only. For instance, if v is a slice, it returns the size of the slice descriptor, not the size of the memory referenced by the slice.

type [ArbitraryType](#)

```
type ArbitraryType int
```

ArbitraryType is here for the purposes of documentation only and is not actually part of the unsafe package. It represents the type of an arbitrary Go expression.

type [Pointer](#)

```
type Pointer *ArbitraryType
```

Pointer represents a pointer to an arbitrary type. There are three special operations available for type Pointer that are not available for other types.

- 1) A pointer value of any type can be converted to a Pointer.
- 2) A Pointer can be converted to a pointer value of any type.
- 3) A uintptr can be converted to a Pointer.
- 4) A Pointer can be converted to a uintptr.

Pointer therefore allows a program to defeat the type system and read and write arbitrary memory. It should be used with extreme care.

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Go Projects

These are external projects and not endorsed or supported by the Go project.

Projects

Submit a Project

Using this form you can submit a project to be included in the list.

Filter by tag: **all** [lib](#) [app](#) [tool](#) [cgo](#) [wontbuild](#)

Build Tools

- [GG](#) - A build tool for Go in Go tool
- [GVM](#) - GVM provides an interface to manage Go versions. lib
- [Gocheck](#) - Rich test framework with suites, fixtures, assertions, good error reporting, etc tool
- [SCons Go Tools](#) - A collection of builders that makes it easy to compile Go projects in SCons. tool
- [chimp](#) - Library to provide the "Go" language with automation tools. Can be used to write build scripts, deploy scripts, etc tool
- [fileembed-go](#) - This is a command-line utility to take a number of source files, and embed them into a Go package. tool
- [gb](#) - A(nother) build tool for go, with an emphasis on multi-package projects. tool
- [go-runner](#) - a simple runner for .go programs tool wontbuild
- [goam](#) - A simple project build tool for Go tool
- [gobbler](#) - Go build tool to build and test advanced multi-package projects with little configuration tool wontbuild
- [gobuild](#) - Automatic build tool aiming to replace Makefiles for simple Go projects tool
- [gobuild-fork](#) - A fork of gobuild tool wontbuild
- [godag](#) - A frontend to the Go compiler collection tool
- [goscons](#) - Another set of SCons builders for Go. tool
- [gotgo](#) - An experimental preprocessor to implement 'generics' tool wontbuild
- [makengo](#) - A build tool similar to Ruby's rake tool

Caching

- [gocache](#) - gocache app wontbuild

- [gomemcached](#) - A memcached server in go app

Command-line Option Parsers

- [ReadFlags](#) - Read flag values from text files lib
- [argcfg](#) - Use reflection to populate fields in a struct from command line arguments lib
- [gnuflag](#) - GNU-compatible flag parsing; substantially compatible with flag. lib
- [go-options](#) - A command line parsing library for Go lib
- [goopt](#) - a getopt clone to parse command-line flags lib
- [optarg](#) - An easy to use GNU-style command-line argument parsing with full validation and nice usage information. lib
- [optparse-go](#) - parses command-lines lib
- [opts.go](#) - lightweight POSIX- and GNU- style option parsing lib
- [pflag](#) - Drop-in replacement for Go's flag package, implementing POSIX/GNU-style --flags. lib

Command-line Tools

- [GoPasswordCreator](#) - A small tool, which creates random passwords tool
- [cron.go](#) - A small cron job system to handle scheduled tasks, such as optimizing databases or kicking idle users from chat. lib
- [gich](#) - A cross platform which utility written in Go tool
- [goblin](#) - a set of Plan9/unix utilities re-written in Go app
- [gocreate](#) - Command line utility that create files from templates. tool
- [jsonpp](#) - A fast command line JSON pretty printer. lib
- [tecla](#) - Command-line editing library lib

Compression

- [dgoz](#) - LZO bindings lib
- [go-lz4](#) - Port of LZ4 lossless compression algorithm to Go. lib
- [go-lzss](#) - Implementation of LZSS compression algorithm in Go lib
- [go-sevenz](#) - Package sevenzip implements access to 7-zip archives (wraps C interface of LZMA SDK) lib
- [lzma](#) - compress/lzma package for Go lib
- [snappy-go](#) - Google's Snappy compression algorithm in Go lib

- [yenc](#) - yenc decoder package lib
- [zip](#) - a library for reading ZIP archives. lib

Console User Interface

- [DevTodo2](#) - A small command-line per-project task list manager. tool
- [ansiterm](#) - pkg to drive text-only consoles that respond to ANSI escape sequences
- [curses.go](#) - GO binding for NCurses lib
- [g5t](#) - Gettext for Go, stripped down to the bare metal lib
- [getpass](#) - Password prompting from console lib
- [go-stfl](#) - a thin wrapper around STFL, an ncurses-based widget toolkit lib
- [go-term](#) - Wrapper and utilities related to Unix terminal lib
- [gockel](#) - a Twitter client for text terminals tool
- [gocurse](#) - Go bindings for NCurses cgo wontbuild
- [goncurses](#) - An ncurses library, including the form, menu and panel extensions lib
- [igo](#) - A simple interactive Go interpreter built on exp/eval with some readline refinements lib wontbuild
- [inline](#) - Library for line editing lib
- [oh](#) - A Unix shell written in Go app
- [pty](#) - obtain pseudo-terminal devices lib
- [tattr-go](#) - Thin wrapper around termios structure lib
- [termios](#) - Terminal support lib
- [termon](#) - Easy terminal-control-interface for Go. lib

Cryptography

- [BLAKE-256](#) - Go implementation of BLAKE-256 hash function lib
- [GoKoblitz](#) - An ECDSA library for Go supporting Koblitz curves (such as secp256k1) lib
- [GoSkein](#) - Implementation of Skein hash and Threefish crypto for Go lib
- [bcrypt](#) - Blowfish password hashing lib
- [dkeyczar](#) - Go port of Google'e Keyczar cryptography library lib
- [dkrcrypt](#) - Korean block ciphers: SEED and HIGHT lib
- [dskipjack](#) - Go implementation of the SKIPJACK encryption algorithm lib
- [go-rabbit](#) - Go implementation of Rabbit encryption algorithm lib
- [go-salsa20](#) - Go implementation of Salsa20 encryption algorithm lib

- [gopam](#) - PAM application API wrapper lib
- [ketama.go](#) - libketama-style consistent hashing lib
- [kindi](#) - encryption command line tool app
- [passwordhash](#) - Package passwordhash implements safe password hashing and comparison. lib
- [pbkdf2.go](#) - PBKDF2 password hashing lib
- [scpu](#) - SSH2 client application lib
- [scrypt](#) - Go implementation of Colin Percival's scrypt key derivation function lib
- [ssh.go](#) - SSH2 Client library lib

Data Structures

- [FSM](#) - Finite State Machine lib
- [GoLLRB](#) - A Left-Leaning Red-Black (LLRB) implementation of 2-3 balanced binary search trees in Google Go lib
- [Gokogiri](#) - A lightweight libxml wrapper library lib
- [Picugen](#) - A general-purpose hash/checksum digest generator. lib
- [Sortutil](#) - Nested, case-insensitive, and reverse sorting for Go. lib
- [asyncwr](#) - Asynchronous, non-blocking, wrapper for io.Writer lib
- [bigendian](#) - binary parsing and printing lib
- [btree](#) - Package btree implements B-trees with fixed size keys, <http://en.wikipedia.org/wiki/Btree> lib
- [deepcopy](#) - Make deep copies of data structures lib wontbuild
- [dgobloom](#) - A Bloom Filter implementation lib
- [epochdate](#) - Compact dates stored as days since the Unix epoch lib
- [go-avltree](#) - AVL tree (Adel'son-Vel'skii & Landis) with indexing added lib
- [go-darts](#) - Double-ARray Trie System for golang lib
- [go-extractor](#) - Go wrapper for GNU libextractor lib
- [go-maps](#) - Go maps generalized to interfaces lib
- [go-priority-queue](#) - An easy to use heap implementation with a conventional priority queue interface. lib
- [go-stree](#) - A segment tree implementation for range queries on intervals lib
- [gobson](#) - BSON (de)serializer lib
- [gohash](#) - A simple linked-list hashtable that implements sets and maps lib
- [goop](#) - Dynamic object-oriented programming support for Go lib
- [gopqueue](#) - Priority queue at top of container/heap lib
- [gorbt](#) - A simple red black tree lib
- [gotoc](#) - A protocol buffer compiler written in Go lib

- [goxml](#) - A thin wrapper around libxml2 lib
- [gringo](#) - A minimalist queue implemented using a stripped-down lock-free ringbuffer lib
- [libgob](#) - A low level library for generating gobs from other languages lib
- [seq](#) - Functional containers, sequential and concurrent lib
- [skip.go](#) - Fast ordered maps using skip lists lib
- [skiplist](#) - A skip list implementation. Highly customizable and easy to use. lib
- [sortutil](#) - Utilities supplemental to the Go standard "sort" package lib
- [timsort](#) - Fast, stable sort, uses external comparator lib
- [tribool](#) - Ternary (tree-valued) logic for Go lib
- [vcard](#) - Reading and writing vcard file in go. Implementation of RFC 2425 (A MIME Content-Type for Directory Information) and RFC 2426 (vCard MIME Directory Profile). lib
- [xlsx](#) - A library to help with extracting data from Microsoft Office Excel XLSX files. lib
- [xmlm](#) - Advanced XML marshalling/unmarshalling lib

Databases and Storage

- [CodeSearch](#) - Index and perform regex searches over large bodies of source code app
- [DBGo](#) - A light-weight relational flat-file database engine. app
- [Disky](#) - Home-grown, disk-backed key-value store tool
- [Go-Redis](#) - Client and Connectors for Redis key-value store lib
- [GoMySQL](#) - MySQL library for Go lib
- [GoMySQL-Client-Library](#) - Go MySQL Client Library lib
- [MyMySQL](#) - MySQL Client API written entirely in Go. lib
- [Optimus Cache Prime](#) - Smart cache preloader for websites with XML sitemaps. app
- [PostgreSQL bindings](#) - PostgreSQL bindings cgo
- [Radix](#) - Asynchronous Redis client lib
- [Tideland CGL Redis](#) - Powerful Redis client with pub/sub support. lib
- [Tideland Redis Database Client](#) - Simple but powerful client for the Redis key/value (and more) database. lib
- [Weed File System](#) - fast distributed key-file store app
- [cabinet](#) - Kyoto Cabinet bindings for go lib
- [cass](#) - Cassandra Client Lib lib
- [cdb.go](#) - Create and read cdb ("constant database") files lib

- [couch-go](#) - newer maintained CouchDB database binding lib
- [fswatch](#) - mac utility for watching the file system for changes tool
- [go datamapper](#) - ActiveRecord-like database wrapper lib
- [go-cache](#) - An in-memory key:value store/cache (similar to Memcached) library for Go, suitable for single-machine applications. app
- [go-db](#) - A generic database API lib
- [go-db-oracle](#) - GO interface to Oracle DB lib
- [go-dbd-mysql](#) - A MySQL driver implementation for go-dbi lib
- [go-dbi](#) - A database abstraction API in the spirit of Perl DBI et al lib
- [go-mongo](#) - a driver for MongoDB lib
- [go-mysql](#) - MySQL wrapper for Go cgo
- [go-mysql-driver](#) - A lightweight and fast MySQL-Driver for Go's database/sql package lib
- [go-notify](#) - GO bindings for the libnotify lib
- [go-odbc](#) - ODBC Driver for Go lib
- [go-pgsql](#) - A PostgreSQL client library for Go lib
- [go-sphinx](#) - A sphinx client package for Go, for full text search. lib
- [go-sqlite3](#) - Access SQLite3 databases from Go cgo
- [go-wikiparse](#) - mediawiki dump parser for working with wikipedia data tool
- [go.fsevents](#) - path event lib for mac lib
- [gocask](#) - Key-value store inspired by Riak Bitcask. Can be used as pure go implementation of dbm and other kv-stores. lib
- [gocouch](#) - a CouchDB client library lib
- [godbc](#) - Go ODBC Interface using unixODBC by Benoy R Nair cgo
- [godis](#) - Simple client for Redis lib
- [gofluid](#) - FluidDB client library for Go lib
- [gofluiddb](#) - A lightweight wrapper around the FluidDB API for clients written in Go lib
- [gographite](#) - statsd server in go (for feeding data to graphite) app
- [gokabinet](#) - Go bindings for Kyoto Cabinet DBM implementation lib
- [gomemcache](#) - a memcached client lib
- [gomongo](#) - driver for MongoDB lib
- [goprotodb](#) - A binding to Berkeley DB storing records encoded as Protocol Buffers. lib
- [goriak](#) - Database bindings for Riak lib
- [goriak](#) - Database driver for riak database (project homepage is now on bitbucket.org) lib
- [gorp](#) - SQL mapper for Go lib
- [gosqlite](#) - a trivial SQLite binding for Go. lib

- [gosqlite \(forked\)](#) - A fork of gosqlite lib
- [gosqlite3](#) - Go Interface for SQLite3 cgo
- [gotyrant](#) - A Go wrapper for tokyo tyrant cgo
- [hdfs.go](#) - go bindings for libhdfs lib
- [levigo](#) - levigo provides the ability to create and access LevelDB databases. lib
- [libmysqlgo](#) - Another wrapper for the MySQL C API cgo
- [mgo](#) - Rich MongoDB driver for Go lib
- [mgo \(package\)](#) - Rich MongoDB driver for Go (package reference for the Packages tab, please keep the other link since it has docs) lib
- [mysql-connector-go](#) - implements MySQL wire protocol lib
- [mysqlgo](#) - MySQL bindings cgo
- [persival](#) - Programatic, persistent, pseudo key-value storage app
- [pgsql.go](#) - PostgreSQL high-level client library wrapper lib
- [pq.go](#) - A pure Go Postgres driver lib
- [vitess](#) - Scaling MySQL databases for the web app

Development Tools

- [deifdef](#) - Removes #ifdef/#endif blocks from code based on a set of #defines tool
- [glib](#) - Bindings for GLib type system lib
- [gonew](#) - A tool to create new Go projects tool
- [gonow](#) - Tool to run Go scripts tool
- [gorun](#) - Enables Go source files to be used as scripts. tool
- [hamcrest](#) - Hamcrest matchers (and Asserters) for runtime check and JUnit-like testing tool lib
- [hsandbox](#) - Tool for quick experimentation with Go snippets tool
- [liccor](#) - A tool for updating license headers in Go source files tool
- [liteide](#) - An go auto build tools and qt-based ide for Go tool
- [trace](#) - a simple debug tracing tool

Distributed/Grid Computing

- [Cloud-Backups](#) - Small Go utilities to backup data from the cloud, to the cloud app
- [malus](#) - A Kademlia-compatible DHT(Distributed Hash Table) written in Go app

Documentation

- [Mango](#) - Automatically generate unix man pages from Go sources app
- [pkgdoc](#) - Pkgdoc generates HTML documentation for a Go package using a user specified template. tool
- [redoc](#) - Commands documentation for Redis tool

Editors

- [Go conTEXT](#) - Highlighter plugin for the conTEXT editor lib
- [Google Go for Idea](#) - Google Go language plugin for IntelliJ IDEA
- [goclipse](#) - An Eclipse-based IDE for Go. tool
- [gofinder](#) - (code) search tool for acme tool
- [tabby](#) - Source code editor app

Encodings

- [mimemagic](#) - Detect mime-types automatically based on file contents with no external dependencies lib

GUIs and Widget Toolkits

- [go-fltk](#) - FLTK2 GUI toolkit bindings for Go lib
- [go-gtk](#) - Bindings for GTK cgo
- [go-iup](#) - Bindings for Iup, a cross-platform native widget GUI toolkit lib
- [goosurface](#) - An interface to Cairo via GTK cgo
- [mdtwm](#) - Tiling window manager for X app
- [termbox](#) - A minimalist alternative to ncurses to build terminal-based user interfaces cgo

Games

- [ChessBuddy](#) - Play chess with Go, HTML5, WebSockets and random strangers! app
- [FlexBot](#) - A flexible bot for the Spring RTS engine implementing the 'lobby protocol' app
- [Gongo](#) - A program written in Go that plays Go app
- [godoku](#) - Go Sudoku Solver - example of "share by communicating" tool

- [gospeccy](#) - A ZX Spectrum 48k Emulator app
- [gotris](#) - A classic tetris game written in Go app
- [teratogen](#) - A rogue-like game using SDL app

Go Implementations

- [Express Go](#) - Interpreted Go implementation for Windows app

Graphics and Audio

- [Arclight](#) - Arclight is a tool for rendering images app
- [Go-OpenGL](#) - Go bindings for OpenGL cgo
- [Go-SDL](#) - Go bindings for SDL cgo
- [GoGL](#) - OpenGL binding generator lib
- [GoMacDraw](#) - A mac implementation of go.wde lib
- [Goop](#) - Audio synthesizer engine tool
- [Winhello](#) - An example Windows GUI hello world application app
- [allergro](#) - basic wrapper for the Allegro library cgo
- [alsa](#) - alsa is a Go wrapper package for C alsa library. lib
- [baukasten](#) - A modular game library. lib
- [blend](#) - Image processing library and rendering toolkit for Go. lib
- [bmp.go](#) - package for encoding/decoding Windows BMP files lib
- [chart](#) - Library to generate common chart (pie, bar, strip, scatter, histogram) in different output formats. lib
- [draw2d](#) - This package provide an API to draw 2d geometrical form on images. This library is largely inspired by postscript, cairo, HTML5 canvas. lib
- [freetype-go](#) - a Go implementation of FreeType lib
- [gmf](#) - Go Media Framework - a binding to ffmpeg to bring Media Processing to Google Go lib
- [go-gd](#) - Go bingings for GD lib
- [go-gnuplot](#) - go bindings for GNUPlot lib
- [go-graph](#) - Generic graph library for Go. lib
- [go-gtk3](#) - gtk3 bindings for go lib
- [go-heatmap](#) - A toolkit for making heatmaps lib
- [go-openal](#) - Experimental OpenAL bindings for Go cgo
- [go-opencl](#) - A go wrapper to the OpenCL heterogeneous parallel programming library lib

- [go-vlc](#) - Go bindings for libVLC lib
- [go.uik](#) - A UI kit for Go, in Go. lib
- [go.wde](#) - A windowing/drawing/event interface lib
- [gocairo](#) - Golang wrapper for cairo graphics library cgo
- [gofax](#) - CCITT fax library lib
- [goray](#) - Raytracer written in Go, based on Yafaray app
- [gosdl](#) - Go wrapper for SDL cgo
- [goxscr](#) - Go rewrites of xscreensaver ports app
- [gst](#) - Go bindings for GStreamer lib
- [hgui](#) - Gui toolkit based on http and gtk-webkit. lib
- [iascii](#) - retro-ASCII image encoder tool
- [id3tag](#) - id3tag is a Go wrapper around C libid3tag library. lib
- [math3d](#) - A linear algebra package optimized for OpenGL lib
- [ogg](#) - Go wrapper for C libogg library. lib cgo
- [pdfreader](#) - a library to read the contents of PDF files lib
- [portaudio](#) - A Go binding to PortAudio lib
- [postscript-go](#) - Postscript go implementation that uses draw2d to generate images. lib
- [pulsego](#) - Go binding for PulseAudio cgo
- [smallpt.go](#) - A port of the smallpt global illumination renderer to Go app
- [starfish](#) - A simple Go graphics and user input library, built on SDL lib
- [svgo](#) - a library for creating and outputting SVG lib
- [window](#) - Optimized moving window for real-time data lib
- [wingo](#) - A fully-featured window manager written in Go. app
- [wxGo](#) - Go Wrapper for the wxWidgets GUI lib
- [x-go-binding](#) - bindings for the X windowing system lib

Instant Messaging

- [Go-IRC-Client-Library](#) - blah lib
- [GoTY](#) - "Go Troll Yourself", minimalist client IRC library lib
- [calculon](#) - IRC bot with support for runtime (un)loadable modules and configurable via a web interface lib
- [go-bot](#) - (aka rndbot) - An irc-bot that executes Go code sent to it and print its output app
- [goirc](#) - event-based stateful IRC client framework lib
- [irc.go](#) - Go IRC bot framework lib

Mathematics

- [Bitcoin Calculator](#) - Bitcoin Mining, Power and Profitability Calculator, data fetching, json, scheduled tasks lib
- [GoStats](#) - Descriptive statistics and linear regression for Go lib
- [Tideland Go Numerical Library](#) - Helpful numerical types and functions. lib
- [bayesian](#) - Naive Bayesian Classification for Golang lib
- [blas](#) - Go implementation of BLAS (Basic Linear Algebra Subprograms) lib
- [cartconvert](#) - cartography functions for the Go programming language lib
- [ellipsoid](#) - ellipsoid.go performs latitude and longitude calculations on the surface of an ellipsoid. lib
- [geom](#) - 2d geometry. lib
- [go-fftw](#) - Go bindings for FFTW - The Fastest Fourier Transform in the West lib
- [go-fn](#) - Special functions that would not fit in "math" pkg lib
- [go-gt](#) - Graph theory algorithms lib
- [go-humanize](#) - Formatting numbers for humans. lib
- [go.matrix](#) - a linear algebra package (please remove gomatrix - i am its author and this is the same code) lib
- [gochipmunk](#) - Go bindings to the Chipmunk Physics library. lib
- [gocomplex](#) - a complex number library lib
- [gofrac](#) - A fractions library for go lib
- [gogmp](#) - GMP bindings for Go cgo
- [gomatrix](#) - a linear algebra package lib
- [imath](#) - Fast integer-only math routines lib
- [mathutil](#) - Package mathutil provides utilities supplementing the standard 'math' and 'rand' packages. lib
- [polyclip.go](#) - Go implementation of algorithm for Boolean operations on 2D polygons lib

Misc

- [ArBit](#) - Automated Bitcoin arbitrage trading program. tool
- [CGRates](#) - Rating system designed to be used in telecom carriers world
- [Go-PhysicsFS](#) - Go bindings for the PhysicsFS archive-access abstraction library. lib
- [Go-fuse](#) - Library to write FUSE filesystems in Go lib
- [GoLCS](#) - Solve Longest Common Sequence problem in go lib

- [GoTS](#) - a commandline tool for www.tinysong.com app
- [Gotgo](#) - A Go preprocessor that provides an implementation of generics tool
- [Hranoprovod](#) - Command-line calorie tracking tool
- [Perlito](#) - An implementation of a subset of Perl 6 with a Go (and other languages) backend tool
- [Tideland CGL Monitoring](#) - Flexible monitoring of your application lib
- [Tideland Common Go Library](#) - Many helpful packages for the daily Go development. lib
- [Tideland Event-Driven Cell Architecture](#) - Package for the development of event-driven architectures. lib
- [UbuntuTranslator](#) - a simple but useful translator for Ubuntu. app
- [atexit](#) - Simple atexit library lib
- [bencoding](#) - Go implementation of the bencoding protocol used by bittorrent lib
- [bio-go](#) - Basic bioinformatics functions for the Go language. lib
- [cpu](#) - A Go package that reports processor topology lib
- [dbus-go](#) - D-Bus Go library lib
- [desktop](#) - Open file/uri with default application (cross platform) lib
- [dump](#) - An utility that dumps Go variables, similar to PHP's var_dump tool
- [dupfinder](#) - Simple program that finds duplicate files app
- [errforce](#) - Utilities for working effectively with Go errors and UNIX errors lib
- [faker](#) - Generate fake data, names, text, addresses, etc. tool
- [fnv](#) - Go implementation for the Fowler-Noll-Vo hash function lib
- [functional](#) - Functional programming library including a lazy list implementation and some of the most usual functions. lib
- [fungo](#) - A library to help write 2D games in Go lib
- [go-amiando](#) - Wrapper for the Amiando event management API lib
- [go-bit](#) - An efficient and comprehensive bitset implementation with utility bit functions. lib
- [go-eco](#) - Functions for use in ecology lib
- [go-erx](#) - Extended error reporting library lib
- [go-ext](#) - Small utility library for Go lib
- [go-hyphenator](#) - A TeX-style hyphenation package. lib
- [go-idn](#) - a project to bring IDN support to Go, feature compatible with libidn lib
- [go-papi](#) - Go interface to the PAPI performance API lib
- [go-pkg-lastfm](#) - A library to access the entire Last.fm 2.0 webservice API, including the authenticated services lib

- [go-pkg-mpd](#) - A library to access the MPD music daemon lib
- [go-pkg-mtp](#) - Bindings for the libmtp implementation of the Media Transfer Protocol to interact with media devices like MP3 players cgo
- [go-pkg-njb](#) - Bindings for the libnjb (Nomad Juke Box) that interacts with older MP3 players cgo
- [go-pkg-xmlx](#) - Extension to the standard Go XML package. Maintains a node tree that allows forward/backwards browser and exposes some simple single/multi-node search functions lib
- [go-qrand](#) - Go client for quantum random bit generator service at random.irb.hr lib
- [go-repl](#) - A Go REPL; builds up a source .go file over time, compiles it for output tool
- [go-semvar](#) - Semantic versions (see <http://semver.org>) lib
- [go-taskstats](#) - Go interface for Linux taskstats lib
- [go-translate](#) - Google Language Translate library lib
- [go-uuid](#) - Universal Unique IDentifier generator and parser lib
- [go.pcsc-lite](#) - Go wrapper for pcsc-lite lib
- [goNI488](#) - A Go wrapper around National Instruments NI488.2 General Purpose Interface Bus (GPIB) driver. lib
- [goPromise](#) - Scheme-like delayed evaluation for Go lib
- [goagain](#) - Zero-downtime restarts in Go lib
- [goconf](#) - a configuration file parser lib
- [goconfig](#) - Configuration file parser for Go lib
- [gocsv](#) - Library for CSV parsing and emitting lib
- [goga](#) - A genetic algorithm framework lib
- [gogobject](#) - GObject-introspection based bindings generator lib
- [golife](#) - Implementation of Game of Life for command line app
- [gomagic](#) - Libmagic bindings cgo
- [gommap](#) - gommap enables Go programs to directly work with memory mapped files and devices in a very efficient way lib
- [goneuro](#) - Go driver for NeuroSky devices. lib
- [goplan9](#) - libraries for interacting with Plan 9 lib
- [goraphing](#) - A tool to generate a simple graph data structures from JSON data files app
- [goskirt](#) - Upskirt markdown library bindings for Go lib
- [gotaskqueue](#) - With gotaskqueue, a program could define several tasks and execute them separately at specific time points. lib
- [gotimer](#) - A simple way to time a function lib
- [gotweet](#) - A simple Twitter command line client app

- [gouuid](#) - Pure Go UUID v3, 4 and 5 generator compatible with RFC4122 lib
- [hasher](#) - Library to compute hash in user account passwords lib
- [iolaus-go](#) - A Go implementation of the 'iolaus' distributed version control system app
- [koans](#) - programming koans for go
- [lineup](#) - A minimalistic message queue server app
- [log4go](#) - Go logging package akin to log4j lib
- [magic](#) - wrapper around libmagic lib
- [mimeparse](#) - Simple library to handle mime-types Go lib
- [mitigation](#) - Package mitigation provides the possibility to prevent damage caused by bugs or exploits. lib
- [passwd](#) - A parser for the /etc/passwd file lib
- [primegen.go](#) - Sieve of Atkin prime number generator
- [seelog](#) - powerful and easy-to-learn logging framework that provides functionality for flexible dispatching, filtering, and formatting lib
- [selenium](#) - Selenium client tool
- [symutils](#) - Various tools and libraries to handle symbolic links lib
- [tamias](#) - a port of Chipmunk Physics lib
- [tideland-kmr](#) - Tideland Knowledge Management and Retrieval is a wiki app
- [trie](#) - A Trie structure implementation for Go, using Unicode runes as keys. Includes a customization for TeX-style hyphenation tries. lib
- [xplor](#) - Files tree browser for p9p acme app

Music

- [gompd](#) - A client interface for the MPD (Music Player Daemon) app

Networking

- [Go Ajax](#) - Go Ajax is a JSON-RPC implementation designed to create AJAX powered websites. lib
- [GoAWS](#) - Library for many AWS services (S3, SQS, EC2, etc) lib
- [GoPOP3](#) - Implements the POP3 protocol as specified in RFC 1939 lib
- [GoRTP](#) - RTP / RTCP stack implementation for Go lib
- [HTTP JSON-RPC](#) - An implementation of HTTP JSON-RPC protocol for Go lib

- [Skynet](#) - Skynet is distributed mesh of processes designed for highly scalable API type service provision. lib
- [Tonika](#) - Secure social networking platform app
- [Uniqush](#) - A free and open source software which provides a unified push service for server-side notification to apps on mobile devices. lib
- [cascadeauth](#) - Squid authenticator that consults multiple sources app
- [dmrigo](#) - Library for with Hadoop Streaming map/reduce lib
- [dns](#) - A DNS library in Go lib
- [doozerd](#) - A consistent distributed data store app
- [dyndnsd](#) - a configurable dyndns client tool
- [eventsourc](#) - Server-sent events for net/http server. lib
- [freeport](#) - Find a free port on the system. lib
- [ftp.go](#) - FTP client for Google Go language lib
- [ftp4go](#) - An FTP client for Go, started as a port of the standard Python FTP client library lib
- [gearman-go](#) - A native implementation for Gearman API with Go. lib
- [go-curl](#) - libcurl binding that supports go func callbacks lib
- [go-dbus](#) - A library to connect to the D-bus messaging system lib
- [go-icap](#) - ICAP (Internet Content Adaptation Protocol) server library lib
- [go-imap](#) - IMAP client library lib
- [go-irc](#) - Simple IRC client library lib
- [go-mail](#) - Email utilities including RFC822 messages and Google Mail defaults. lib
- [go-msgpack](#) - MsgPack library for Go, with pack/unpack and net/rpc codec support lib
- [go-nagios](#) - Library for writing Nagios plugins lib
- [go-nntp](#) - An NNTP client and server library for go lib
- [go-router](#) - implementation of remote channel communication lib
- [go-rpcgen](#) - ProtoBuf RPC binding generator for net/rpc and AppEngine lib
- [go-socket.io](#) - A Socket.IO backend implementation written in Go lib
- [go-xmlrpc](#) - Simple XML-RPC client/server library lib
- [go-xmpp](#) - XMPP client library lib
- [go9](#) - an implementation of the 9P distributed file system protocol lib
- [go9p](#) - 9p protocol implementation in Go lib
- [goauth](#) - A library for header-based OAuth over HTTP or HTTPS. lib
- [gobeanstalk](#) - Go Beanstalkd client library lib
- [gobir](#) - Extensible IRC bot with channel administration, seen support, and go documentation querying app
- [gocluster](#) - implementation of a clustering heuristic using a particle swarm

- optimization technique lib
- [godloader](#) - Collection of download tools that tries to follow the Unix philosophy app
- [godns](#) - A more complete DNS package lib
- [godns2](#) - A simple DNS resolution library with fine control over the messages lib
- [godwulf](#) - Gopher server written in Go app
- [goexmpp](#) - XMPP client implementation lib
- [goftp](#) - A FTP client library lib
- [gogammu](#) - Library for sending and receiving SMS lib
- [gonetbench](#) - Simple TCP benchmarking tool tool
- [gonoip](#) - No-IP client lib
- [gopcap](#) - A simple wrapper around libpcap cgo
- [goprotobuf](#) - the Go implementation of Google's Protocol Buffers lib
- [gorobot](#) - a modular IRC bot app
- [gosndfile](#) - Go binding for libsndfile lib
- [gostomp](#) - implementation of STOMP (Streaming Text Orientated Messaging Protocol) lib
- [gozmq](#) - Go Bindings for 0mq (zeromq/zmq) lib
- [grong](#) - Small authoritative DNS name server app
- [handlersocket-go](#) - Go native library to connect to HandlerSocket interface of InnoDB tables lib
- [jaid](#) - Just Another IRC Daemon app
- [kafka.go](#) - Producer & Consumer for the Kafka messaging system lib
- [ldap](#) - Basic LDAP v3 functionality for the GO programming language. lib
- [mdns](#) - Multicast DNS library for Go lib
- [netsnail](#) - A low-bandwidth simulator app
- [remotize](#) - A remotize package and command that helps remotizing methods without having to chaneg their signatures for rpc lib
- [replican-sync](#) - An rsync algorithm implementation in Go lib
- [rs232](#) - Serial interface for those of us who still have modems (or arduinos) lib
- [statsd.go](#) - Client library for statsd lib
- [stompngo](#) - A Stomp 1.1 Compliant Client lib
- [stompngo_examples](#) - Examples for stompngo. lib

Other Random Toys, Experiments and Example Code

- [go-crazy](#) - An experimental source-to-source compiler for go tool

- [go-hashmap](#) - A hash table in pure go as an experiment in Go performance app
- [gochat](#) - A 'stupid' chat server written in Go app
- [goconc](#) - A collection of useful concurrency idioms and functions for Go, compiled app
- [goplay](#) - A bunch of random small programs in Go app
- [lifegame-on-golang](#) - Game of Life in Go app
- [linear](#) - Playing around with the linear algebra app
- [project euler in go](#) - Solutions to Project Euler in Go also app
- [shadergo](#) - shader test using golang app

P2P and File Sharing

- [Taipei-Torrent](#) - Another BT client app
- [ed2kcrawler](#) - eDonkey2000 link crawler app
- [gobit](#) - Bittorrent Client in Go app
- [gop2p](#) - A simple p2p app to learn Go app
- [wgo](#) - A simple BitTorrent client based in part on the Taipei-Torrent and gobit code app

Programming

- [GoSpec](#) - a BDD framework lib
- [go-clang](#) - cgo bindings to the C-API of libclang lib
- [go-galib](#) - a library of Genetic Algorithms lib
- [go-intset](#) - a library to work with bounded sets of integers, including multiple alternative implementations lib
- [go-parse](#) - a Parsec-like parsing library lib
- [go-stringio](#) - implementation of the various file I/O interfaces using memory buffers instead of real files lib
- [godeferred](#) - port of jsdeferred: <http://cho45.stfuawsc.com/jsdeferred/> lib
- [gosets](#) - implementation of set types lib
- [gospecify](#) - another BDD framework lib
- [gowizard](#) - Tool to create skeleton of Go projects tool
- [iterutils](#) - functions from Python's itertools module lib

Source Code Management

- [hggofmt](#) - A Mercurial/hg extension with a hook to *gofmt* changed files automatically before a commit tool

Strings and Text

- [Black Friday](#) - A markdown processor lib
- [GoXML](#) - A basic libxml wrapper for Go lib cgo
- [Mahonia](#) - Character-set conversion library in Go lib
- [NTemplate](#) - Nested Templates lib
- [csvutil](#) - A heavy duty CSV reading and writing library. lib
- [dgohash](#) - Collection of string hashing functions, including Murmur3 and others lib
- [go-charset](#) - Conversion between character sets. Native Go. lib
- [go-guess](#) - Go wrapper for libguess lib
- [go-migemo](#) - migemo extension for go (Japanese incremental text search) lib
- [go-substrs](#) - A container for substrings resultant from pattern matching lib
- [go.stringmetrics](#) - String distance metrics implemented in Go lib
- [goini](#) - A go library to parse INI files. lib
- [gosphinx](#) - A Go client interface to the Sphinx standalone full-text search engine app
- [goyaml](#) - A port of LibYAML to Go lib
- [gpKMP](#) - String-matching in Golang using the Knuth  CMorris  CPratt algorithm lib
- [iconv-go](#) - iconv wrapper with Reader and Writer lib
- [inflect](#) - Word inflection library (similar to Ruby ActiveSupport::Inflector). Singularize(), Pluralize(), Underscore() etc. lib
- [kasia.go](#) - Templating system for HTML and other text documents lib
- [kview](#) - Simple wrapper for kasia.go templates. It helps to modularize content of a website lib
- [mail.go](#) - Parse email messages lib
- [neste](#) - Extended version of Go's template package for generating textual output from nested templates. lib
- [peg](#) - Parsing Expression Grammer Parser lib
- [pretty.go](#) - Pretty-printing for go values lib
- [rubex](#) - A simple regular expression library that supports Ruby's regex syntax. It is faster than Regexp. lib
- [scanner](#) - A text scanner that parses primitive types, analogous to Java's `java.util.Scanner` or C's `scanf(3)` lib

- [sre2](#) - RE2 in Go lib
- [strogonoff](#) - Stenography with Go lib
- [strutil](#) - Package strutil collects utils supplemental to the standard strings package. lib
- [uctricks](#) - Some silly text transformations via Unicode lib

Tag Generators

- [egotags](#) - ETags generator tool
- [gotags](#) - Generate a tags file for the Go Programming Language in the format used tool
- [tago](#) - Emacs TAGS generator for Go source tool

Testing

- [Tideland CGL Asserts](#) - Make asserts during testing and inside of your applications lib
- [assert](#) - helper functions for the built-in 'testing' package lib
- [go2xunit](#) - Convert "go test -v" output to xunit XML output tool
- [gomock](#) - a mocking framework for Go. tool

Virtual Machines and Languages

- [Gelo](#) - Extensible, embeddable interpreter app
- [GoForth](#) - A simple Forth parser app
- [GoLightly](#) - A flexible and lightweight virtual machine with runtime-configurable instruction set app
- [JavaScriptCore](#) - This is a wrapper for WebKit's javascript engine for Go. lib
- [RubyGoLightly](#) - An experimental port of TinyRb to Go app
- [The erGo? Compiler](#) - An independent implementation of the Go language. app
- [brainfuck](#) - A brainfuck virtual machine in Go app
- [forego](#) - Forth virtual machine tool
- [go-python](#) - go bindings for CPython C-API lib
- [go-scheme](#) - Scheme implementation in Go app
- [goheader](#) - Tool for translating C type declarations into its Go equivalent lib
- [golemon](#) - A port of the Lemon parser-generator lib

- [goll1e](#) - An LL(1) parser generator for the Go programming language. app
- [golua](#) - Go wrapper for LUA's C API cgo
- [golua-fork](#) - A fork of GoLua that works on current releases of Go lib
- [gotcl](#) - Tcl interpreter in Go lib
- [ngaro](#) - A ngaro virtual machine to run retroForth images app
- [prescript](#) - An experimental PostScript-like scripting language app
- [turing](#) - BF virtual machine more appropriate for school settings. lib

Web Applications

- [Digestw](#) - A Web Application - Twitter's Timeline Digest app
- [GoURLShortener](#) - A frontend for the <http://is.gd/> URL shortener app
- [J♦♦Vai Tarde](#) - Unfollows monitoring for Twitter app
- [fourohfourfound](#) - A fallback HTTP server that may redirect requests with runtime configurable redirections app
- [goals-calendar](#) - A web-based Seinfeld calendar implemented in Go app
- [goblog](#) - A static blog engine lib
- [goflash](#) - Flash player implementation in Go language app
- [gogallery](#) - simple web server with an emphasis on easily browsing images app
- [goof](#) - A simple http server to exchange files over http (upload/download) app
- [gopages](#) - A php-like web framework that allows embedding Go code in web pages app
- [gopaste](#) - The code that runs the gopaste.org pastebin app
- [goplot](#) - A graphing utility with some curve-fitting features, includes a web interface app
- [gowiki](#) - A simple wiki in Go using web.go and mustache.go app
- [htdigest-go](#) - Command line tool for managing htdigest files lib
- [kurz.go](#) - a url shortener based on web.go and redis app
- [now.go](#) - A simple HTTP-based to-do queue. app
- [sf_server](#) - a tiny send file server and client lib
- [webtf](#) - Web app to graphical visualization of twitter timelines using the HTML5 `<canvas>` tag app

Web Libraries

- [Cascadia](#) - CSS selector library lib

- [GOAuth](#) - OAuth Consumer lib
- [Go-OAuth](#) - OAuth 1.0 client lib
- [GoRest](#) - An extensive configuration(tags) based RESTful style web-services framework. lib
- [GoWeb](#) - Frameworklet that simplifies building API's in Go - has Ruby on Rails style routing lib
- [Goldorak.Go](#) - a web miniframework built using mustache.go, web.go and Go-Redis lib
- [HTML Transform](#) - A CSS selector based html scraping and transformation library lib
- [Kontl](#) - A client for kon.tl's URL shortening service lib
- [OAuth Consumer](#) - OAuth 1.0 consumer implementation lib
- [RSS-Go](#) - RSS and ATOM feed reader package for the Go programming language. lib
- [Stack on Go](#) - Go wrapper for Stack Exchange API lib
- [Tideland CGL Web](#) - Package for RESTful web applications lib
- [Tideland RESTful Web Framework](#) - A foundation for web applications and servers following the REST principles lib
- [Twister](#) - A framework and server for writing web applications. lib
- [app.go](#) - Web framework for google app engine lib
- [authcookie](#) - Package authcookie implements creation and verification of signed authentication cookies. lib
- [bwl](#) - a set of libraries to help build web sites lib
- [captcha](#) - Image and audio captcha generator and server lib
- [ddg](#) - DuckDuckGo API interface lib
- [dgoogauth](#) - Go port of Google's Authenticator library for one-time passwords lib
- [falcore](#) - Modular HTTP server framework lib
- [fcgigo](#) - a FastCGI implementation lib
- [gaerecords](#) - Lightweight wrapper around appengine/datastore providing Active Record and DBO style management of data lib
- [get2ch-go](#) - a library to access the 2channel Japanese web bulletin board lib
- [go-dealmap](#) - Go library for accessing TheDealMap's API lib
- [go-dropbox](#) - API library for dropbox lib
- [go-facebook](#) - Go implementations of facebook APIs. lib
- [go-fastweb](#) - aims to be a simple, small and clean MVC framework for go lib
- [go-flickr](#) - A wrapper for Flickr's API lib
- [go-gravatar](#) - Wrapper for the Gravatar API lib

- [go-gzip-file-server](#) - A net.http.Handler similar to FileServer that serves gzipped content lib
- [go-http-auth](#) - HTTP Basic and HTTP Digest authentication lib
- [go-libGeoIP](#) - GO Lib GeoIP API for Maxmind lib
- [go-pkg-rss](#) - a packages that reads RSS and Atom feeds lib
- [go-rss](#) - Simple RSS parser, tested with Wordpress feeds. lib
- [go-start](#) - A high level web-framework for Go lib
- [go-tripit](#) - Go API library for the TripIt web services lib
- [go-twitter](#) - another Twitter client lib
- [go-twitter-oauth](#) - a simple Twitter client (supports OAuth) lib
- [go-urlshortener](#) - interface to google's urlshorten API lib
- [go-webproject](#) - Modular web application framework and app server tool
- [godom](#) - a library that implements a small, non-compliant subset of the W3C DOM Core lib
- [gofastcgi](#) - another FastCGI implementation lib
- [gohaml](#) - An implementation of the popular XHTML Abstraction Markup Language using the Go language. lib
- [gojwt](#) - Json Web Tokens for Go lib
- [gomesh](#) - A simple HTML decoration library. lib
- [googtrans](#) - unofficial go bindings for Google Translate API v2 lib
- [gorilla](#) - Go web toolkit lib
- [gorouter](#) - Simple router for go to process url variables lib
- [goscribble](#) - An MPD Audioscrobble lib
- [goweb](#) - Lightweight web framework for Go providing Ruby on Rails style routing lib
- [htmlfiller](#) - Fills in html forms with default values and errors a la Ian Bicking's htmlfill for Python lib
- [http-gonsole](#) - Speak HTTP like a local. (the simple, intuitive HTTP console, golang version) lib
- [httplib.go](#) - 'Low level' client HTTP library that provides keep-alive connections and generic requests lib
- [justintv](#) - Justin.tv REST API with oauth lib
- [mango](#) - Mango is a modular web-application framework for Go, inspired by Rack, and PEP333. lib
- [mustache.go](#) - an implementation of the Mustache template language lib
- [passwordreset](#) - Creation and verification of secure tokens useful for implementation of "reset forgotten password" feature in web applications. lib
- [postmark](#) - Access postmark API from Go lib

- [pusher.go](#) - HTTP Server Push module for the standard http package lib
- [rest.go](#) - Library to simplify implementation of REST servers and clients. lib
- [rest.go \(forked\)](#) - forked rest.go for improvements and REST consistency lib
- [rest2go](#) - Based on rest.go, forked for improvements and REST consistency lib
- [robotstxt](#) - The robots.txt exclusion protocol implementation. Allows to parse and query robots.txt file. lib
- [routes.go](#) - http routing API similar to expressjs or sinatra lib
- [seshcookie](#) - A web session library inspired by Beaker lib
- [web](#) - A web framework with views for constructing web page, supports FCGI, CGI. lib
- [web.go](#) - a simple framework to write webapps lib
- [webdriver](#) - WebDriver (Selenium) client lib
- [webtestutil](#) - Web and HTTP functional testing utilities. Includes Gorilla testing support. lib
- [wfd](#) - Simple web framework designed for and written in go. Works with other languages as well, but not as well. lib
- [xmldom](#) - a library that implements a small, non-compliant subset of the W3C DOM Core lib
- [xsrfToken](#) - A package for generating and validating tokens used in preventing XSRF attacks. lib

Windows

- [gform](#) - An easy to use Windows GUI toolkit for Go lib
- [go-Windows-begin](#) - for the absolute Windows-Go beginner
- [go-ole](#) - win32 ole implementation for golang lib
- [w32](#) - Windows API wrapper for Go. lib
- [walk](#) - "Windows Application Library Kit" for the Go Programming Language lib

Source file

src/pkg/archive/tar/common.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package tar implements access to tar archives.
6 // It aims to cover most of the variations, including those
7 // by GNU and BSD tars.
8 //
9 // References:
10 //   http://www.freebsd.org/cgi/man.cgi?query=tar&sektion=5
11 //   http://www.gnu.org/software/tar/manual/html_node/Standards
12 package tar
13
14 import "time"
15
16 const (
17     blockSize = 512
18
19     // Types
20     TypeReg          = '0' // regular file
21     TypeRegA        = '\x00' // regular file
22     TypeLink        = '1' // hard link
23     TypeSymlink     = '2' // symbolic link
24     TypeChar        = '3' // character device node
25     TypeBlock       = '4' // block device node
26     TypeDir         = '5' // directory
27     TypeFifo        = '6' // fifo node
28     TypeCont        = '7' // reserved
29     TypeXHeader     = 'x' // extended header
30     TypeXGlobalHeader = 'g' // global extended header
31 )
32
33 // A Header represents a single header in a tar archive.
34 // Some fields may not be populated.
35 type Header struct {
36     Name      string // name of header file entry
37     Mode      int64  // permission and mode bits
38     Uid       int    // user id of owner
39     Gid       int    // group id of owner
40     Size      int64  // length in bytes
41     ModTime   time.Time // modified time
```

```

42     Typeflag    byte    // type of header entry
43     Linkname   string   // target name of link
44     Uname      string   // user name of owner
45     Gname      string   // group name of owner
46     Devmajor   int64    // major number of character or
47     Devminor   int64    // minor number of character or
48     AccessTime time.Time // access time
49     ChangeTime time.Time // status change time
50 }
51
52 var zeroBlock = make([]byte, blockSize)
53
54 // POSIX specifies a sum of the unsigned byte values, but th
55 // We compute and return both.
56 func checksum(header []byte) (unsigned int64, signed int64)
57     for i := 0; i < len(header); i++ {
58         if i == 148 {
59             // The chksum field (header[148:156]
60             unsigned += ' ' * 8
61             signed += ' ' * 8
62             i += 7
63             continue
64         }
65         unsigned += int64(header[i])
66         signed += int64(int8(header[i]))
67     }
68     return
69 }
70
71 type slicer []byte
72
73 func (sp *slicer) next(n int) (b []byte) {
74     s := *sp
75     b, *sp = s[0:n], s[n:]
76     return
77 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/archive/tar/reader.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tar
6
7 // TODO(dsymonds):
8 //   - pax extensions
9
10 import (
11     "bytes"
12     "errors"
13     "io"
14     "io/ioutil"
15     "os"
16     "strconv"
17     "time"
18 )
19
20 var (
21     ErrHeader = errors.New("archive/tar: invalid tar hea
22 )
23
24 // A Reader provides sequential access to the contents of a
25 // A tar archive consists of a sequence of files.
26 // The Next method advances to the next file in the archive
27 // and then it can be treated as an io.Reader to access the
28 //
29 // Example:
30 //     tr := tar.NewReader(r)
31 //     for {
32 //         hdr, err := tr.Next()
33 //         if err == io.EOF {
34 //             // end of tar archive
35 //             break
36 //         }
37 //         if err != nil {
38 //             // handle error
39 //         }
40 //         io.Copy(data, tr)
41 //     }
```

```

42 type Reader struct {
43     r io.Reader
44     err error
45     nb int64 // number of unread bytes for current file
46     pad int64 // amount of padding (ignored) after curre
47 }
48
49 // NewReader creates a new Reader reading from r.
50 func NewReader(r io.Reader) *Reader { return &Reader{r: r} }
51
52 // Next advances to the next entry in the tar archive.
53 func (tr *Reader) Next() (*Header, error) {
54     var hdr *Header
55     if tr.err == nil {
56         tr.skipUnread()
57     }
58     if tr.err == nil {
59         hdr = tr.readHeader()
60     }
61     return hdr, tr.err
62 }
63
64 // Parse bytes as a NUL-terminated C-style string.
65 // If a NUL byte is not found then the whole slice is return
66 func cString(b []byte) string {
67     n := 0
68     for n < len(b) && b[n] != 0 {
69         n++
70     }
71     return string(b[0:n])
72 }
73
74 func (tr *Reader) octal(b []byte) int64 {
75     // Removing leading spaces.
76     for len(b) > 0 && b[0] == ' ' {
77         b = b[1:]
78     }
79     // Removing trailing NULs and spaces.
80     for len(b) > 0 && (b[len(b)-1] == ' ' || b[len(b)-1]
81         b = b[0 : len(b)-1]
82     }
83     x, err := strconv.ParseUint(cString(b), 8, 64)
84     if err != nil {
85         tr.err = err
86     }
87     return int64(x)
88 }
89
90 // Skip any unread bytes in the existing file entry, as well
91 func (tr *Reader) skipUnread() {

```

```

92         nr := tr.nb + tr.pad // number of bytes to skip
93         tr.nb, tr.pad = 0, 0
94         if sr, ok := tr.r.(io.Seeker); ok {
95             if _, err := sr.Seek(nr, os.SEEK_CUR); err =
96                 return
97             }
98         }
99         _, tr.err = io.CopyN(ioutil.Discard, tr.r, nr)
100     }
101
102     func (tr *Reader) verifyChecksum(header []byte) bool {
103         if tr.err != nil {
104             return false
105         }
106
107         given := tr.octal(header[148:156])
108         unsigned, signed := checksum(header)
109         return given == unsigned || given == signed
110     }
111
112     func (tr *Reader) readHeader() *Header {
113         header := make([]byte, blockSize)
114         if _, tr.err = io.ReadFull(tr.r, header); tr.err !=
115             return nil
116         }
117
118         // Two blocks of zero bytes marks the end of the arc
119         if bytes.Equal(header, zeroBlock[0:blockSize]) {
120             if _, tr.err = io.ReadFull(tr.r, header); tr
121                 return nil
122             }
123         if bytes.Equal(header, zeroBlock[0:blockSize]
124             tr.err = io.EOF
125         } else {
126             tr.err = ErrHeader // zero block and
127         }
128         return nil
129     }
130
131     if !tr.verifyChecksum(header) {
132         tr.err = ErrHeader
133         return nil
134     }
135
136     // Unpack
137     hdr := new(Header)
138     s := slicer(header)
139
140     hdr.Name = cString(s.next(100))

```

```

141     hdr.Mode = tr.octal(s.next(8))
142     hdr.Uid = int(tr.octal(s.next(8)))
143     hdr.Gid = int(tr.octal(s.next(8)))
144     hdr.Size = tr.octal(s.next(12))
145     hdr.ModTime = time.Unix(tr.octal(s.next(12)), 0)
146     s.next(8) // chksum
147     hdr.Typeflag = s.next(1)[0]
148     hdr.Linkname = cString(s.next(100))
149
150     // The remainder of the header depends on the value
151     // The original (v7) version of tar had no explicit
152     // so its magic bytes, like the rest of the block, a
153     magic := string(s.next(8)) // contains version field
154     var format string
155     switch magic {
156     case "ustar\x0000": // POSIX tar (1003.1-1988)
157         if string(header[508:512]) == "tar\x00" {
158             format = "star"
159         } else {
160             format = "posix"
161         }
162     case "ustar \x00": // old GNU tar
163         format = "gnu"
164     }
165
166     switch format {
167     case "posix", "gnu", "star":
168         hdr.Uname = cString(s.next(32))
169         hdr.Gname = cString(s.next(32))
170         devmajor := s.next(8)
171         devminor := s.next(8)
172         if hdr.Typeflag == TypeChar || hdr.Typeflag
173             hdr.Devmajor = tr.octal(devmajor)
174             hdr.Devminor = tr.octal(devminor)
175         }
176     var prefix string
177     switch format {
178     case "posix", "gnu":
179         prefix = cString(s.next(155))
180     case "star":
181         prefix = cString(s.next(131))
182         hdr.AccessTime = time.Unix(tr.octal(
183             hdr.ChangeTime = time.Unix(tr.octal(
184         }
185     if len(prefix) > 0 {
186         hdr.Name = prefix + "/" + hdr.Name
187     }
188 }
189

```

```

190         if tr.err != nil {
191             tr.err = ErrHeader
192             return nil
193         }
194
195         // Maximum value of hdr.Size is 64 GB (12 octal digi
196         // so there's no risk of int64 overflowing.
197         tr.nb = int64(hdr.Size)
198         tr.pad = -tr.nb & (blockSize - 1) // blockSize is a
199
200         return hdr
201     }
202
203     // Read reads from the current entry in the tar archive.
204     // It returns 0, io.EOF when it reaches the end of that entr
205     // until Next is called to advance to the next entry.
206     func (tr *Reader) Read(b []byte) (n int, err error) {
207         if tr.nb == 0 {
208             // file consumed
209             return 0, io.EOF
210         }
211
212         if int64(len(b)) > tr.nb {
213             b = b[0:tr.nb]
214         }
215         n, err = tr.r.Read(b)
216         tr.nb -= int64(n)
217
218         if err == io.EOF && tr.nb > 0 {
219             err = io.ErrUnexpectedEOF
220         }
221         tr.err = err
222         return
223     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/archive/tar/writer.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tar
6
7 // TODO(dsymonds):
8 // - catch more errors (no first header, etc.)
9
10 import (
11     "errors"
12     "fmt"
13     "io"
14     "strconv"
15 )
16
17 var (
18     ErrWriteTooLong      = errors.New("archive/tar: write
19     ErrFieldTooLong     = errors.New("archive/tar: header
20     ErrWriteAfterClose = errors.New("archive/tar: write
21 )
22
23 // A Writer provides sequential writing of a tar archive in
24 // A tar archive consists of a sequence of files.
25 // Call WriteHeader to begin a new file, and then call Write
26 // writing at most hdr.Size bytes in total.
27 //
28 // Example:
29 //     tw := tar.NewWriter(w)
30 //     hdr := new(Header)
31 //     hdr.Size = length of data in bytes
32 //     // populate other hdr fields as desired
33 //     if err := tw.WriteHeader(hdr); err != nil {
34 //         // handle error
35 //     }
36 //     io.Copy(tw, data)
37 //     tw.Close()
38 type Writer struct {
39     w      io.Writer
40     err    error
41     nb    int64 // number of unwritten bytes for cu
```

```

42         pad          int64 // amount of padding to write after
43         closed       bool
44         usedBinary  bool // whether the binary numeric field
45     }
46
47     // NewWriter creates a new Writer writing to w.
48     func NewWriter(w io.Writer) *Writer { return &Writer{w: w} }
49
50     // Flush finishes writing the current file (optional).
51     func (tw *Writer) Flush() error {
52         if tw.nb > 0 {
53             tw.err = fmt.Errorf("archive/tar: missed wri
54             return tw.err
55         }
56
57         n := tw.nb + tw.pad
58         for n > 0 && tw.err == nil {
59             nr := n
60             if nr > blockSize {
61                 nr = blockSize
62             }
63             var nw int
64             nw, tw.err = tw.w.Write(zeroBlock[0:nr])
65             n -= int64(nw)
66         }
67         tw.nb = 0
68         tw.pad = 0
69         return tw.err
70     }
71
72     // Write s into b, terminating it with a NUL if there is room
73     func (tw *Writer) cString(b []byte, s string) {
74         if len(s) > len(b) {
75             if tw.err == nil {
76                 tw.err = ErrFieldTooLong
77             }
78             return
79         }
80         copy(b, s)
81         if len(s) < len(b) {
82             b[len(s)] = 0
83         }
84     }
85
86     // Encode x as an octal ASCII string and write it into b with
87     func (tw *Writer) octal(b []byte, x int64) {
88         s := strconv.FormatInt(x, 8)
89         // leading zeros, but leave room for a NUL.
90         for len(s)+1 < len(b) {
91             s = "0" + s

```

```

92         }
93         tw.cString(b, s)
94     }
95
96     // Write x into b, either as octal or as binary (GNUTar/star
97     func (tw *Writer) numeric(b []byte, x int64) {
98         // Try octal first.
99         s := strconv.FormatInt(x, 8)
100        if len(s) < len(b) {
101            tw.octal(b, x)
102            return
103        }
104        // Too big: use binary (big-endian).
105        tw.usedBinary = true
106        for i := len(b) - 1; x > 0 && i >= 0; i-- {
107            b[i] = byte(x)
108            x >>= 8
109        }
110        b[0] |= 0x80 // highest bit indicates binary format
111    }
112
113    // WriteHeader writes hdr and prepares to accept the file's
114    // WriteHeader calls Flush if it is not the first header.
115    // Calling after a Close will return ErrWriteAfterClose.
116    func (tw *Writer) WriteHeader(hdr *Header) error {
117        if tw.closed {
118            return ErrWriteAfterClose
119        }
120        if tw.err == nil {
121            tw.Flush()
122        }
123        if tw.err != nil {
124            return tw.err
125        }
126
127        tw.nb = int64(hdr.Size)
128        tw.pad = -tw.nb & (blockSize - 1) // blockSize is a
129
130        header := make([]byte, blockSize)
131        s := slicer(header)
132
133        // TODO(dsymonds): handle names longer than 100 char
134        copy(s.next(100), []byte(hdr.Name))
135
136        tw.octal(s.next(8), hdr.Mode)           // 100:10
137        tw.numeric(s.next(8), int64(hdr.Uid))   // 108:11
138        tw.numeric(s.next(8), int64(hdr.Gid))   // 116:12
139        tw.numeric(s.next(12), hdr.Size)        // 124:13
140        tw.numeric(s.next(12), hdr.ModTime.Unix()) // 136:14

```

```

141         s.next(8) // chksum
142         s.next(1)[0] = hdr.Typeflag // 156:15
143         tw.cString(s.next(100), hdr.Linkname) // linkna
144         copy(s.next(8), []byte("ustar\x0000")) // 257:26
145         tw.cString(s.next(32), hdr.Uname) // 265:29
146         tw.cString(s.next(32), hdr.Gname) // 297:32
147         tw.numeric(s.next(8), hdr.Devmajor) // 329:33
148         tw.numeric(s.next(8), hdr.Devminor) // 337:34
149
150         // Use the GNU magic instead of POSIX magic if we us
151         if tw.usedBinary {
152             copy(header[257:265], []byte("ustar \x00"))
153         }
154
155         // The chksum field is terminated by a NUL and a spa
156         // This is different from the other octal fields.
157         chksum, _ := checksum(header)
158         tw.octal(header[148:155], chksum)
159         header[155] = ' '
160
161         if tw.err != nil {
162             // problem with header; probably integer too
163             return tw.err
164         }
165
166         _, tw.err = tw.w.Write(header)
167
168         return tw.err
169     }
170
171     // Write writes to the current entry in the tar archive.
172     // Write returns the error ErrWriteTooLong if more than
173     // hdr.Size bytes are written after WriteHeader.
174     func (tw *Writer) Write(b []byte) (n int, err error) {
175         if tw.closed {
176             err = ErrWriteTooLong
177             return
178         }
179         overwrite := false
180         if int64(len(b)) > tw.nb {
181             b = b[0:tw.nb]
182             overwrite = true
183         }
184         n, err = tw.w.Write(b)
185         tw.nb -= int64(n)
186         if err == nil && overwrite {
187             err = ErrWriteTooLong
188             return
189         }

```

```

190         tw.err = err
191         return
192     }
193
194     // Close closes the tar archive, flushing any unwritten
195     // data to the underlying writer.
196     func (tw *Writer) Close() error {
197         if tw.err != nil || tw.closed {
198             return tw.err
199         }
200         tw.Flush()
201         tw.closed = true
202         if tw.err != nil {
203             return tw.err
204         }
205
206         // trailer: two zero blocks
207         for i := 0; i < 2; i++ {
208             _, tw.err = tw.w.Write(zeroBlock)
209             if tw.err != nil {
210                 break
211             }
212         }
213         return tw.err
214     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/archive/zip/reader.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package zip
6
7 import (
8     "bufio"
9     "compress/flate"
10    "encoding/binary"
11    "errors"
12    "hash"
13    "hash/crc32"
14    "io"
15    "io/ioutil"
16    "os"
17 )
18
19 var (
20     ErrFormat      = errors.New("zip: not a valid zip file")
21     ErrAlgorithm   = errors.New("zip: unsupported compress")
22     ErrChecksum    = errors.New("zip: checksum error")
23 )
24
25 type Reader struct {
26     r      io.ReaderAt
27     File   []*File
28     Comment string
29 }
30
31 type ReadCloser struct {
32     f *os.File
33     Reader
34 }
35
36 type File struct {
37     FileHeader
38     zipr      io.ReaderAt
39     zipsize   int64
40     headerOffset int64
41 }
```

```

42
43 func (f *File) hasDataDescriptor() bool {
44     return f.Flags&0x8 != 0
45 }
46
47 // OpenReader will open the Zip file specified by name and r
48 func OpenReader(name string) (*ReadCloser, error) {
49     f, err := os.Open(name)
50     if err != nil {
51         return nil, err
52     }
53     fi, err := f.Stat()
54     if err != nil {
55         f.Close()
56         return nil, err
57     }
58     r := new(ReadCloser)
59     if err := r.init(f, fi.Size()); err != nil {
60         f.Close()
61         return nil, err
62     }
63     r.f = f
64     return r, nil
65 }
66
67 // NewReader returns a new Reader reading from r, which is a
68 // have the given size in bytes.
69 func NewReader(r io.ReaderAt, size int64) (*Reader, error) {
70     zr := new(Reader)
71     if err := zr.init(r, size); err != nil {
72         return nil, err
73     }
74     return zr, nil
75 }
76
77 func (z *Reader) init(r io.ReaderAt, size int64) error {
78     end, err := readDirectoryEnd(r, size)
79     if err != nil {
80         return err
81     }
82     z.r = r
83     z.File = make([]*File, 0, end.directoryRecords)
84     z.Comment = end.comment
85     rs := io.NewSectionReader(r, 0, size)
86     if _, err = rs.Seek(int64(end.directoryOffset), os.S
87         return err
88     }
89     buf := bufio.NewReader(rs)
90
91     // The count of files inside a zip is truncated to f

```

```

92     // Gloss over this by reading headers until we encou
93     // a bad one, and then only report a ErrFormat or Un
94     // the file count modulo 65536 is incorrect.
95     for {
96         f := &File{zipr: r, zipsize: size}
97         err = readDirectoryHeader(f, buf)
98         if err == ErrFormat || err == io.ErrUnexpect
99             break
100    }
101    if err != nil {
102        return err
103    }
104    z.File = append(z.File, f)
105 }
106 if uint16(len(z.File)) != end.directoryRecords {
107     // Return the readDirectoryHeader error if w
108     // the wrong number of directory entries.
109     return err
110 }
111 return nil
112 }
113
114 // Close closes the Zip file, rendering it unusable for I/O.
115 func (rc *ReadCloser) Close() error {
116     return rc.f.Close()
117 }
118
119 // Open returns a ReadCloser that provides access to the Fil
120 // Multiple files may be read concurrently.
121 func (f *File) Open() (rc io.ReadCloser, err error) {
122     bodyOffset, err := f.findBodyOffset()
123     if err != nil {
124         return
125     }
126     size := int64(f.CompressedSize)
127     r := io.NewSectionReader(f.zipr, f.headerOffset+body
128     switch f.Method {
129     case Store: // (no compression)
130         rc = ioutil.NopCloser(r)
131     case Deflate:
132         rc = flate.NewReader(r)
133     default:
134         err = ErrAlgorithm
135         return
136     }
137     var desr io.Reader
138     if f.hasDataDescriptor() {
139         desr = io.NewSectionReader(f.zipr, f.headerC
140     }

```

```

141         rc = &checksumReader{rc, crc32.NewIEEE(), f, descr, n
142         return
143     }
144
145     type checksumReader struct {
146         rc    io.ReadCloser
147         hash hash.Hash32
148         f     *File
149         descr io.Reader // if non-nil, where to read the data
150         err   error    // sticky error
151     }
152
153     func (r *checksumReader) Read(b []byte) (n int, err error) {
154         if r.err != nil {
155             return 0, r.err
156         }
157         n, err = r.rc.Read(b)
158         r.hash.Write(b[:n])
159         if err == nil {
160             return
161         }
162         if err == io.EOF {
163             if r.descr != nil {
164                 if err1 := readDataDescriptor(r.descr
165                     err = err1
166                 } else if r.hash.Sum32() != r.f.CRC3
167                     err = ErrChecksum
168                 }
169             } else {
170                 // If there's not a data descriptor,
171                 // the CRC32 of what we've read agai
172                 // or TOC's CRC32, if it seems like
173                 if r.f.CRC32 != 0 && r.hash.Sum32()
174                     err = ErrChecksum
175                 }
176             }
177         }
178         r.err = err
179         return
180     }
181
182     func (r *checksumReader) Close() error { return r.rc.Close()
183
184     // findBodyOffset does the minimum work to verify the file h
185     // and returns the file body offset.
186     func (f *File) findBodyOffset() (int64, error) {
187         r := io.NewSectionReader(f.zipr, f.headerOffset, f.z
188         var buf [fileHeaderLen]byte
189         if _, err := io.ReadFull(r, buf[:]); err != nil {

```

```

190         return 0, err
191     }
192     b := readBuf(buf[:])
193     if sig := b.uint32(); sig != fileHeaderSignature {
194         return 0, ErrFormat
195     }
196     b = b[22:] // skip over most of the header
197     filenameLen := int(b.uint16())
198     extraLen := int(b.uint16())
199     return int64(fileHeaderLen + filenameLen + extraLen)
200 }
201
202 // readDirectoryHeader attempts to read a directory header f
203 // It returns io.ErrUnexpectedEOF if it cannot read a comple
204 // and ErrFormat if it doesn't find a valid header signature
205 func readDirectoryHeader(f *File, r io.Reader) error {
206     var buf [directoryHeaderLen]byte
207     if _, err := io.ReadFull(r, buf[:]); err != nil {
208         return err
209     }
210     b := readBuf(buf[:])
211     if sig := b.uint32(); sig != directoryHeaderSignatur
212         return ErrFormat
213     }
214     f.CreatorVersion = b.uint16()
215     f.ReaderVersion = b.uint16()
216     f.Flags = b.uint16()
217     f.Method = b.uint16()
218     f.ModifiedTime = b.uint16()
219     f.ModifiedDate = b.uint16()
220     f.CRC32 = b.uint32()
221     f.CompressedSize = b.uint32()
222     f.UncompressedSize = b.uint32()
223     filenameLen := int(b.uint16())
224     extraLen := int(b.uint16())
225     commentLen := int(b.uint16())
226     b = b[4:] // skipped start disk number and internal
227     f.ExternalAttrs = b.uint32()
228     f.headerOffset = int64(b.uint32())
229     d := make([]byte, filenameLen+extraLen+commentLen)
230     if _, err := io.ReadFull(r, d); err != nil {
231         return err
232     }
233     f.Name = string(d[:filenameLen])
234     f.Extra = d[filenameLen : filenameLen+extraLen]
235     f.Comment = string(d[filenameLen+extraLen:])
236     return nil
237 }
238
239 func readDataDescriptor(r io.Reader, f *File) error {

```

```

240     var buf [dataDescriptorLen]byte
241
242     // The spec says: "Although not originally assigned
243     // signature, the value 0x08074b50 has commonly been
244     // as a signature value for the data descriptor reco
245     // Implementers should be aware that ZIP files may b
246     // encountered with or without this signature markin
247     // descriptors and should account for either case wh
248     // ZIP files to ensure compatibility."
249     //
250     // dataDescriptorLen includes the size of the signat
251     // first read just those 4 bytes to see if it exists
252     if _, err := io.ReadFull(r, buf[:4]); err != nil {
253         return err
254     }
255     off := 0
256     maybeSig := readBuf(buf[:4])
257     if maybeSig.uint32() != dataDescriptorSignature {
258         // No data descriptor signature. Keep these
259         // bytes.
260         off += 4
261     }
262     if _, err := io.ReadFull(r, buf[off:12]); err != nil
263         return err
264     }
265     b := readBuf(buf[:12])
266     f.CRC32 = b.uint32()
267     f.CompressedSize = b.uint32()
268     f.UncompressedSize = b.uint32()
269     return nil
270 }
271
272 func readDirectoryEnd(r io.ReaderAt, size int64) (dir *direc
273     // look for directoryEndSignature in the last 1k, th
274     var buf []byte
275     for i, bLen := range []int64{1024, 65 * 1024} {
276         if bLen > size {
277             bLen = size
278         }
279         buf = make([]byte, int(bLen))
280         if _, err := r.ReadAt(buf, size-bLen); err !=
281             return nil, err
282         }
283         if p := findSignatureInBlock(buf); p >= 0 {
284             buf = buf[p:]
285             break
286         }
287         if i == 1 || bLen == size {
288             return nil, ErrFormat

```

```

289         }
290     }
291
292     // read header into struct
293     b := readBuf(buf[4:]) // skip signature
294     d := &directoryEnd{
295         diskNbr:          b.uint16(),
296         dirDiskNbr:      b.uint16(),
297         dirRecordsThisDisk: b.uint16(),
298         directoryRecords: b.uint16(),
299         directorySize:    b.uint32(),
300         directoryOffset:  b.uint32(),
301         commentLen:      b.uint16(),
302     }
303     l := int(d.commentLen)
304     if l > len(b) {
305         return nil, errors.New("zip: invalid comment")
306     }
307     d.comment = string(b[:l])
308     return d, nil
309 }
310
311 func findSignatureInBlock(b []byte) int {
312     for i := len(b) - directoryEndLen; i >= 0; i-- {
313         // defined from directoryEndSignature in str
314         if b[i] == 'P' && b[i+1] == 'K' && b[i+2] ==
315             // n is length of comment
316             n := int(b[i+directoryEndLen-2]) | i
317             if n+directoryEndLen+i == len(b) {
318                 return i
319             }
320         }
321     }
322     return -1
323 }
324
325 type readBuf []byte
326
327 func (b *readBuf) uint16() uint16 {
328     v := binary.LittleEndian.Uint16(*b)
329     *b = (*b)[2:]
330     return v
331 }
332
333 func (b *readBuf) uint32() uint32 {
334     v := binary.LittleEndian.Uint32(*b)
335     *b = (*b)[4:]
336     return v
337 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/archive/zip/struct.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6 Package zip provides support for reading and writing ZIP arc
7
8 See: http://www.pkware.com/documents/casestudies/APPNOTE.TXT
9
10 This package does not support ZIP64 or disk spanning.
11 */
12 package zip
13
14 import (
15     "errors"
16     "os"
17     "time"
18 )
19
20 // Compression methods.
21 const (
22     Store    uint16 = 0
23     Deflate uint16 = 8
24 )
25
26 const (
27     fileHeaderSignature      = 0x04034b50
28     directoryHeaderSignature = 0x02014b50
29     directoryEndSignature    = 0x06054b50
30     dataDescriptorSignature  = 0x08074b50 // de-facto st
31     fileHeaderLen            = 30         // + filename
32     directoryHeaderLen       = 46         // + filename
33     directoryEndLen          = 22         // + comment
34     dataDescriptorLen        = 16         // four uint32
35
36     // Constants for the first byte in CreatorVersion
37     creatorFAT    = 0
38     creatorUnix   = 3
39     creatorNTFS   = 11
40     creatorVFAT   = 14
41     creatorMacOSX = 19
```

```

42 )
43
44 type FileHeader struct {
45     Name          string
46     CreatorVersion uint16
47     ReaderVersion uint16
48     Flags         uint16
49     Method        uint16
50     ModifiedTime  uint16 // MS-DOS time
51     ModifiedDate  uint16 // MS-DOS date
52     CRC32         uint32
53     CompressedSize uint32
54     UncompressedSize uint32
55     Extra         []byte
56     ExternalAttrs  uint32 // Meaning depends on Creator
57     Comment       string
58 }
59
60 // FileInfo returns an os.FileInfo for the FileHeader.
61 func (h *FileHeader) FileInfo() os.FileInfo {
62     return headerFileInfo{h}
63 }
64
65 // headerFileInfo implements os.FileInfo.
66 type headerFileInfo struct {
67     fh *FileHeader
68 }
69
70 func (fi headerFileInfo) Name() string      { return fi.fh.Name() }
71 func (fi headerFileInfo) Size() int64      { return int64(fi.fh.UncompressedSize) }
72 func (fi headerFileInfo) IsDir() bool      { return fi.fh.Flags & 0x0001 != 0 }
73 func (fi headerFileInfo) ModTime() time.Time { return fi.fh.ModifiedTime }
74 func (fi headerFileInfo) Mode() os.FileMode { return fi.fh.Flags }
75 func (fi headerFileInfo) Sys() interface{} { return fi.fh }
76
77 // FileInfoHeader creates a partially-populated FileHeader from
78 // os.FileInfo.
79 func FileInfoHeader(fi os.FileInfo) (*FileHeader, error) {
80     size := fi.Size()
81     if size > (1<<32 - 1) {
82         return nil, errors.New("zip: file over 4GB")
83     }
84     fh := &FileHeader{
85         Name:          fi.Name(),
86         UncompressedSize: uint32(size),
87     }
88     fh.SetModTime(fi.ModTime())
89     fh.SetMode(fi.Mode())
90     return fh, nil
91 }

```

```

92
93 type directoryEnd struct {
94     diskNbr          uint16 // unused
95     dirDiskNbr       uint16 // unused
96     dirRecordsThisDisk uint16 // unused
97     directoryRecords uint16
98     directorySize     uint32
99     directoryOffset   uint32 // relative to file
100    commentLen        uint16
101    comment            string
102 }
103
104 // msDosTimeToTime converts an MS-DOS date and time into a t
105 // The resolution is 2s.
106 // See: http://msdn.microsoft.com/en-us/library/ms724247\(v=v
107 func msDosTimeToTime(dosDate, dosTime uint16) time.Time {
108     return time.Date(
109         // date bits 0-4: day of month; 5-8: month;
110         int(dosDate>>9+1980),
111         time.Month(dosDate>>5&0xf),
112         int(dosDate&0x1f),
113
114         // time bits 0-4: second/2; 5-10: minute; 11
115         int(dosTime>>11),
116         int(dosTime>>5&0x3f),
117         int(dosTime&0x1f*2),
118         0, // nanoseconds
119
120         time.UTC,
121     )
122 }
123
124 // timeToMsDosTime converts a time.Time to an MS-DOS date an
125 // The resolution is 2s.
126 // See: http://msdn.microsoft.com/en-us/library/ms724274\(v=v
127 func timeToMsDosTime(t time.Time) (fDate uint16, fTime uint16) {
128     t = t.In(time.UTC)
129     fDate = uint16(t.Day() + int(t.Month())<<5 + (t.Year
130     fTime = uint16(t.Second()/2 + t.Minute())<<5 + t.Hour
131     return
132 }
133
134 // ModTime returns the modification time.
135 // The resolution is 2s.
136 func (h *FileHeader) ModTime() time.Time {
137     return msDosTimeToTime(h.ModifiedDate, h.ModifiedTime)
138 }
139
140 // SetModTime sets the ModifiedTime and ModifiedDate fields

```

```

141 // The resolution is 2s.
142 func (h *FileHeader) SetModTime(t time.Time) {
143     h.ModifiedDate, h.ModifiedTime = timeToMsDosTime(t)
144 }
145
146 const (
147     // Unix constants. The specification doesn't mention
148     // but these seem to be the values agreed on by tool
149     s_IFMT    = 0xf000
150     s_IFSOCK  = 0xc000
151     s_IFLNK   = 0xa000
152     s_IFREG   = 0x8000
153     s_IFBLK   = 0x6000
154     s_IFDIR   = 0x4000
155     s_IFCHR   = 0x2000
156     s_IFIFO   = 0x1000
157     s_ISUID   = 0x800
158     s_ISGID   = 0x400
159     s_ISVTX   = 0x200
160
161     msdosDir      = 0x10
162     msdosReadOnly = 0x01
163 )
164
165 // Mode returns the permission and mode bits for the FileHeader
166 func (h *FileHeader) Mode() (mode os.FileMode) {
167     switch h.CreatorVersion >> 8 {
168     case creatorUnix, creatorMacOSX:
169         mode = unixModeToFileMode(h.ExternalAttrs >>
170     case creatorNTFS, creatorVFAT, creatorFAT:
171         mode = msdosModeToFileMode(h.ExternalAttrs)
172     }
173     if len(h.Name) > 0 && h.Name[len(h.Name)-1] == '/' {
174         mode |= os.ModeDir
175     }
176     return mode
177 }
178
179 // SetMode changes the permission and mode bits for the FileHeader
180 func (h *FileHeader) SetMode(mode os.FileMode) {
181     h.CreatorVersion = h.CreatorVersion&0xff | creatorUnix
182     h.ExternalAttrs = fileModeToUnixMode(mode) << 16
183
184     // set MSDOS attributes too, as the original zip does
185     if mode&os.ModeDir != 0 {
186         h.ExternalAttrs |= msdosDir
187     }
188     if mode&0200 == 0 {
189         h.ExternalAttrs |= msdosReadOnly

```

```

190     }
191 }
192
193 func msdosModeToFileMode(m uint32) (mode os.FileMode) {
194     if m&msdosDir != 0 {
195         mode = os.ModeDir | 0777
196     } else {
197         mode = 0666
198     }
199     if m&msdosReadOnly != 0 {
200         mode &^= 0222
201     }
202     return mode
203 }
204
205 func fileModeToUnixMode(mode os.FileMode) uint32 {
206     var m uint32
207     switch mode & os.ModeType {
208     default:
209         m = s_IFREG
210     case os.ModeDir:
211         m = s_IFDIR
212     case os.ModeSymlink:
213         m = s_IFLNK
214     case os.ModeNamedPipe:
215         m = s_IFIFO
216     case os.ModeSocket:
217         m = s_IFSOCK
218     case os.ModeDevice:
219         if mode&os.ModeCharDevice != 0 {
220             m = s_IFCHR
221         } else {
222             m = s_IFBLK
223         }
224     }
225     if mode&os.ModeSetuid != 0 {
226         m |= s_ISUID
227     }
228     if mode&os.ModeSetgid != 0 {
229         m |= s_ISGID
230     }
231     if mode&os.ModeSticky != 0 {
232         m |= s_ISVTX
233     }
234     return m | uint32(mode&0777)
235 }
236
237 func unixModeToFileMode(m uint32) os.FileMode {
238     mode := os.FileMode(m & 0777)
239     switch m & s_IFMT {

```

```

240     case s_IFBLK:
241         mode |= os.ModeDevice
242     case s_IFCHR:
243         mode |= os.ModeDevice | os.ModeCharDevice
244     case s_IFDIR:
245         mode |= os.ModeDir
246     case s_IFIFO:
247         mode |= os.ModeNamedPipe
248     case s_IFLNK:
249         mode |= os.ModeSymlink
250     case s_IFREG:
251         // nothing to do
252     case s_IFSOCK:
253         mode |= os.ModeSocket
254     }
255     if m&s_ISGID != 0 {
256         mode |= os.ModeSetgid
257     }
258     if m&s_ISUID != 0 {
259         mode |= os.ModeSetuid
260     }
261     if m&s_ISVTX != 0 {
262         mode |= os.ModeSticky
263     }
264     return mode
265 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/archive/zip/writer.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package zip
6
7 import (
8     "bufio"
9     "compress/flate"
10    "encoding/binary"
11    "errors"
12    "hash"
13    "hash/crc32"
14    "io"
15 )
16
17 // TODO(adg): support zip file comments
18 // TODO(adg): support specifying deflate level
19
20 // Writer implements a zip file writer.
21 type Writer struct {
22     cw      *countWriter
23     dir     []*header
24     last    *fileWriter
25     closed  bool
26 }
27
28 type header struct {
29     *FileHeader
30     offset uint32
31 }
32
33 // NewWriter returns a new Writer writing a zip file to w.
34 func NewWriter(w io.Writer) *Writer {
35     return &Writer{cw: &countWriter{w: bufio.NewWriter(w)}
36 }
37
38 // Close finishes writing the zip file by writing the centra
39 // It does not (and can not) close the underlying writer.
40 func (w *Writer) Close() error {
41     if w.last != nil && !w.last.closed {
```

```

42         if err := w.last.close(); err != nil {
43             return err
44         }
45         w.last = nil
46     }
47     if w.closed {
48         return errors.New("zip: writer closed twice")
49     }
50     w.closed = true
51
52     // write central directory
53     start := w.cw.count
54     for _, h := range w.dir {
55         var buf [directoryHeaderLen]byte
56         b := writeBuf(buf[:])
57         b.uint32(uint32(directoryHeaderSignature))
58         b.uint16(h.CreatorVersion)
59         b.uint16(h.ReaderVersion)
60         b.uint16(h.Flags)
61         b.uint16(h.Method)
62         b.uint16(h.ModifiedTime)
63         b.uint16(h.ModifiedDate)
64         b.uint32(h.CRC32)
65         b.uint32(h.CompressedSize)
66         b.uint32(h.UncompressedSize)
67         b.uint16(uint16(len(h.Name)))
68         b.uint16(uint16(len(h.Extra)))
69         b.uint16(uint16(len(h.Comment)))
70         b = b[4:] // skip disk number start and inte
71         b.uint32(h.ExternalAttrs)
72         b.uint32(h.offset)
73         if _, err := w.cw.Write(buf[:]); err != nil
74             return err
75     }
76     if _, err := io.WriteString(w.cw, h.Name); e
77         return err
78     }
79     if _, err := w.cw.Write(h.Extra); err != nil
80         return err
81     }
82     if _, err := io.WriteString(w.cw, h.Comment)
83         return err
84     }
85 }
86 end := w.cw.count
87
88 // write end record
89 var buf [directoryEndLen]byte
90 b := writeBuf(buf[:])
91 b.uint32(uint32(directoryEndSignature))

```

```

92         b = b[4:] // skip over disk numb
93         b.uint16(uint16(len(w.dir))) // number of entries t
94         b.uint16(uint16(len(w.dir))) // number of entries t
95         b.uint32(uint32(end - start)) // size of directory
96         b.uint32(uint32(start)) // start of directory
97         // skipped size of comment (always zero)
98         if _, err := w.cw.Write(buf[:]); err != nil {
99             return err
100        }
101
102        return w.cw.w.(*bufio.Writer).Flush()
103    }
104
105    // Create adds a file to the zip file using the provided nam
106    // It returns a Writer to which the file contents should be
107    // The file's contents must be written to the io.Writer befo
108    // call to Create, CreateHeader, or Close.
109    func (w *Writer) Create(name string) (io.Writer, error) {
110        header := &FileHeader{
111            Name:    name,
112            Method: Deflate,
113        }
114        return w.CreateHeader(header)
115    }
116
117    // CreateHeader adds a file to the zip file using the provid
118    // for the file metadata.
119    // It returns a Writer to which the file contents should be
120    // The file's contents must be written to the io.Writer befo
121    // call to Create, CreateHeader, or Close.
122    func (w *Writer) CreateHeader(fh *FileHeader) (io.Writer, er
123        if w.last != nil && !w.last.closed {
124            if err := w.last.close(); err != nil {
125                return nil, err
126            }
127        }
128
129        fh.Flags |= 0x8 // we will write a data descriptor
130        fh.CreatorVersion = fh.CreatorVersion & 0xff00 | 0x14
131        fh.ReaderVersion = 0x14
132
133        fw := &fileWriter{
134            zipw:    w.cw,
135            compCount: &countWriter{w: w.cw},
136            crc32:    crc32.NewIEEE(),
137        }
138        switch fh.Method {
139        case Store:
140            fw.comp = nopCloser{fw.compCount}

```

```

141     case Deflate:
142         var err error
143         fw.comp, err = flate.NewWriter(fw.compCount,
144             if err != nil {
145                 return nil, err
146             }
147     default:
148         return nil, ErrAlgorithm
149     }
150     fw.rawCount = &countWriter{w: fw.comp}
151
152     h := &header{
153         FileHeader: fh,
154         offset:      uint32(w.cw.count),
155     }
156     w.dir = append(w.dir, h)
157     fw.header = h
158
159     if err := writeHeader(w.cw, fh); err != nil {
160         return nil, err
161     }
162
163     w.last = fw
164     return fw, nil
165 }
166
167 func writeHeader(w io.Writer, h *FileHeader) error {
168     var buf [fileHeaderLen]byte
169     b := writeBuf(buf[:])
170     b.uint32(uint32(fileHeaderSignature))
171     b.uint16(h.ReaderVersion)
172     b.uint16(h.Flags)
173     b.uint16(h.Method)
174     b.uint16(h.ModifiedTime)
175     b.uint16(h.ModifiedDate)
176     b.uint32(h.CRC32)
177     b.uint32(h.CompressedSize)
178     b.uint32(h.UncompressedSize)
179     b.uint16(uint16(len(h.Name)))
180     b.uint16(uint16(len(h.Extra)))
181     if _, err := w.Write(buf[:]); err != nil {
182         return err
183     }
184     if _, err := io.WriteString(w, h.Name); err != nil {
185         return err
186     }
187     _, err := w.Write(h.Extra)
188     return err
189 }

```

```

190
191 type fileWriter struct {
192     *header
193     zipw      io.Writer
194     rawCount  *countWriter
195     comp      io.WriteCloser
196     compCount *countWriter
197     crc32     hash.Hash32
198     closed    bool
199 }
200
201 func (w *fileWriter) Write(p []byte) (int, error) {
202     if w.closed {
203         return 0, errors.New("zip: write to closed f
204     }
205     w.crc32.Write(p)
206     return w.rawCount.Write(p)
207 }
208
209 func (w *fileWriter) close() error {
210     if w.closed {
211         return errors.New("zip: file closed twice")
212     }
213     w.closed = true
214     if err := w.comp.Close(); err != nil {
215         return err
216     }
217
218     // update FileHeader
219     fh := w.header.FileHeader
220     fh.CRC32 = w.crc32.Sum32()
221     fh.CompressedSize = uint32(w.compCount.count)
222     fh.UncompressedSize = uint32(w.rawCount.count)
223
224     // write data descriptor
225     var buf [dataDescriptorLen]byte
226     b := writeBuf(buf[:])
227     b.uint32(dataDescriptorSignature) // de-facto standa
228     b.uint32(fh.CRC32)
229     b.uint32(fh.CompressedSize)
230     b.uint32(fh.UncompressedSize)
231     _, err := w.zipw.Write(buf[:])
232     return err
233 }
234
235 type countWriter struct {
236     w      io.Writer
237     count int64
238 }
239

```

```
240 func (w *countWriter) Write(p []byte) (int, error) {
241     n, err := w.w.Write(p)
242     w.count += int64(n)
243     return n, err
244 }
245
246 type nopCloser struct {
247     io.Writer
248 }
249
250 func (w nopCloser) Close() error {
251     return nil
252 }
253
254 type writeBuf []byte
255
256 func (b *writeBuf) uint16(v uint16) {
257     binary.LittleEndian.PutUint16(*b, v)
258     *b = (*b)[2:]
259 }
260
261 func (b *writeBuf) uint32(v uint32) {
262     binary.LittleEndian.PutUint32(*b, v)
263     *b = (*b)[4:]
264 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/bufio/bufio.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package bufio implements buffered I/O. It wraps an io.Re
6 // object, creating another object (Reader or Writer) that a
7 // the interface but provides buffering and some help for te
8 package bufio
9
10 import (
11     "bytes"
12     "errors"
13     "io"
14     "unicode/utf8"
15 )
16
17 const (
18     defaultBufSize = 4096
19 )
20
21 var (
22     ErrInvalidUnreadByte = errors.New("bufio: invalid us
23     ErrInvalidUnreadRune = errors.New("bufio: invalid us
24     ErrBufferFull       = errors.New("bufio: buffer ful
25     ErrNegativeCount    = errors.New("bufio: negative c
26 )
27
28 // Buffered input.
29
30 // Reader implements buffering for an io.Reader object.
31 type Reader struct {
32     buf      []byte
33     rd       io.Reader
34     r, w     int
35     err      error
36     lastByte int
37     lastRuneSize int
38 }
39
40 const minReadBufferSize = 16
41
42 // NewReaderSize returns a new Reader whose buffer has at le
43 // size. If the argument io.Reader is already a Reader with
44 // size, it returns the underlying Reader.
```

```

45 func NewReaderSize(rd io.Reader, size int) *Reader {
46     // Is it already a Reader?
47     b, ok := rd.(*Reader)
48     if ok && len(b.buf) >= size {
49         return b
50     }
51     if size < minReadBufferSize {
52         size = minReadBufferSize
53     }
54     return &Reader{
55         buf:         make([]byte, size),
56         rd:           rd,
57         lastByte:    -1,
58         lastRuneSize: -1,
59     }
60 }
61
62 // NewReader returns a new Reader whose buffer has the default
63 func NewReader(rd io.Reader) *Reader {
64     return NewReaderSize(rd, defaultBufSize)
65 }
66
67 // fill reads a new chunk into the buffer.
68 func (b *Reader) fill() {
69     // Slide existing data to beginning.
70     if b.r > 0 {
71         copy(b.buf, b.buf[b.r:b.w])
72         b.w -= b.r
73         b.r = 0
74     }
75
76     // Read new data.
77     n, e := b.rd.Read(b.buf[b.w:])
78     b.w += n
79     if e != nil {
80         b.err = e
81     }
82 }
83
84 func (b *Reader) readErr() error {
85     err := b.err
86     b.err = nil
87     return err
88 }
89
90 // Peek returns the next n bytes without advancing the reader
91 // being valid at the next read call. If Peek returns fewer
92 // also returns an error explaining why the read is short. T
93 // ErrBufferFull if n is larger than b's buffer size.
94 func (b *Reader) Peek(n int) ([]byte, error) {

```

```

95         if n < 0 {
96             return nil, ErrNegativeCount
97         }
98         if n > len(b.buf) {
99             return nil, ErrBufferFull
100        }
101        for b.w-b.r < n && b.err == nil {
102            b.fill()
103        }
104        m := b.w - b.r
105        if m > n {
106            m = n
107        }
108        var err error
109        if m < n {
110            err = b.readErr()
111            if err == nil {
112                err = ErrBufferFull
113            }
114        }
115        return b.buf[b.r : b.r+m], err
116    }
117
118    // Read reads data into p.
119    // It returns the number of bytes read into p.
120    // It calls Read at most once on the underlying Reader,
121    // hence n may be less than len(p).
122    // At EOF, the count will be zero and err will be io.EOF.
123    func (b *Reader) Read(p []byte) (n int, err error) {
124        n = len(p)
125        if n == 0 {
126            return 0, b.readErr()
127        }
128        if b.w == b.r {
129            if b.err != nil {
130                return 0, b.readErr()
131            }
132            if len(p) >= len(b.buf) {
133                // Large read, empty buffer.
134                // Read directly into p to avoid cop
135                n, b.err = b.rd.Read(p)
136                if n > 0 {
137                    b.lastByte = int(p[n-1])
138                    b.lastRuneSize = -1
139                }
140                return n, b.readErr()
141            }
142            b.fill()
143            if b.w == b.r {

```

```

144             return 0, b.readErr()
145         }
146     }
147
148     if n > b.w-b.r {
149         n = b.w - b.r
150     }
151     copy(p[0:n], b.buf[b.r:])
152     b.r += n
153     b.lastByte = int(b.buf[b.r-1])
154     b.lastRuneSize = -1
155     return n, nil
156 }
157
158 // ReadByte reads and returns a single byte.
159 // If no byte is available, returns an error.
160 func (b *Reader) ReadByte() (c byte, err error) {
161     b.lastRuneSize = -1
162     for b.w == b.r {
163         if b.err != nil {
164             return 0, b.readErr()
165         }
166         b.fill()
167     }
168     c = b.buf[b.r]
169     b.r++
170     b.lastByte = int(c)
171     return c, nil
172 }
173
174 // UnreadByte unreads the last byte. Only the most recently
175 func (b *Reader) UnreadByte() error {
176     b.lastRuneSize = -1
177     if b.r == b.w && b.lastByte >= 0 {
178         b.w = 1
179         b.r = 0
180         b.buf[0] = byte(b.lastByte)
181         b.lastByte = -1
182         return nil
183     }
184     if b.r <= 0 {
185         return ErrInvalidUnreadByte
186     }
187     b.r--
188     b.lastByte = -1
189     return nil
190 }
191
192 // ReadRune reads a single UTF-8 encoded Unicode character a

```

```

193 // rune and its size in bytes. If the encoded rune is invali
194 // and returns unicode.ReplacementChar (U+FFFD) with a size
195 func (b *Reader) ReadRune() (r rune, size int, err error) {
196     for b.r+utf8.UTFMax > b.w && !utf8.FullRune(b.buf[b.
197         b.fill()
198     }
199     b.lastRuneSize = -1
200     if b.r == b.w {
201         return 0, 0, b.readErr()
202     }
203     r, size = rune(b.buf[b.r]), 1
204     if r >= 0x80 {
205         r, size = utf8.DecodeRune(b.buf[b.r:b.w])
206     }
207     b.r += size
208     b.lastByte = int(b.buf[b.r-1])
209     b.lastRuneSize = size
210     return r, size, nil
211 }
212
213 // UnreadRune unreads the last rune. If the most recent rea
214 // the buffer was not a ReadRune, UnreadRune returns an erro
215 // regard it is stricter than UnreadByte, which will unread
216 // from any read operation.)
217 func (b *Reader) UnreadRune() error {
218     if b.lastRuneSize < 0 || b.r == 0 {
219         return ErrInvalidUnreadRune
220     }
221     b.r -= b.lastRuneSize
222     b.lastByte = -1
223     b.lastRuneSize = -1
224     return nil
225 }
226
227 // Buffered returns the number of bytes that can be read fro
228 func (b *Reader) Buffered() int { return b.w - b.r }
229
230 // ReadSlice reads until the first occurrence of delim in th
231 // returning a slice pointing at the bytes in the buffer.
232 // The bytes stop being valid at the next read call.
233 // If ReadSlice encounters an error before finding a delimit
234 // it returns all the data in the buffer and the error itsel
235 // ReadSlice fails with error ErrBufferFull if the buffer fi
236 // Because the data returned from ReadSlice will be overwrit
237 // by the next I/O operation, most clients should use
238 // ReadBytes or ReadString instead.
239 // ReadSlice returns err != nil if and only if line does not
240 func (b *Reader) ReadSlice(delim byte) (line []byte, err err
241     // Look in buffer.
242     if i := bytes.IndexByte(b.buf[b.r:b.w], delim); i >=

```

```

243         line1 := b.buf[b.r : b.r+i+1]
244         b.r += i + 1
245         return line1, nil
246     }
247
248     // Read more into buffer, until buffer fills or we f
249     for {
250         if b.err != nil {
251             line := b.buf[b.r:b.w]
252             b.r = b.w
253             return line, b.readErr()
254         }
255
256         n := b.Buffered()
257         b.fill()
258
259         // Search new part of buffer
260         if i := bytes.IndexByte(b.buf[n:b.w], delim)
261             line := b.buf[0 : n+i+1]
262             b.r = n + i + 1
263             return line, nil
264         }
265
266         // Buffer is full?
267         if b.Buffered() >= len(b.buf) {
268             b.r = b.w
269             return b.buf, ErrBufferFull
270         }
271     }
272     panic("not reached")
273 }
274
275 // ReadLine tries to return a single line, not including the
276 // If the line was too long for the buffer then isPrefix is
277 // beginning of the line is returned. The rest of the line w
278 // from future calls. isPrefix will be false when returning
279 // of the line. The returned buffer is only valid until the
280 // ReadLine. ReadLine either returns a non-nil line or it re
281 // never both.
282 func (b *Reader) ReadLine() (line []byte, isPrefix bool, err
283     line, err = b.ReadSlice('\n')
284     if err == ErrBufferFull {
285         // Handle the case where "\r\n" straddles th
286         if len(line) > 0 && line[len(line)-1] == '\r
287             // Put the '\r' back on buf and drop
288             // Let the next call to ReadLine che
289             if b.r == 0 {
290                 // should be unreachable
291                 panic("bufio: tried to rew

```

```

292         }
293         b.r--
294         line = line[:len(line)-1]
295     }
296     return line, true, nil
297 }
298
299 if len(line) == 0 {
300     if err != nil {
301         line = nil
302     }
303     return
304 }
305 err = nil
306
307 if line[len(line)-1] == '\n' {
308     drop := 1
309     if len(line) > 1 && line[len(line)-2] == '\r'
310         drop = 2
311     }
312     line = line[:len(line)-drop]
313 }
314 return
315 }
316
317 // ReadBytes reads until the first occurrence of delim in th
318 // returning a slice containing the data up to and including
319 // If ReadBytes encounters an error before finding a delimit
320 // it returns the data read before the error and the error i
321 // ReadBytes returns err != nil if and only if the returned
322 // delim.
323 func (b *Reader) ReadBytes(delim byte) (line []byte, err error)
324     // Use ReadSlice to look for array,
325     // accumulating full buffers.
326     var frag []byte
327     var full [][]byte
328     err = nil
329
330     for {
331         var e error
332         frag, e = b.ReadSlice(delim)
333         if e == nil { // got final fragment
334             break
335         }
336         if e != ErrBufferFull { // unexpected error
337             err = e
338             break
339         }
340     }

```

```

341             // Make a copy of the buffer.
342             buf := make([]byte, len(frag))
343             copy(buf, frag)
344             full = append(full, buf)
345         }
346
347         // Allocate new buffer to hold the full pieces and t
348         n := 0
349         for i := range full {
350             n += len(full[i])
351         }
352         n += len(frag)
353
354         // Copy full pieces and fragment in.
355         buf := make([]byte, n)
356         n = 0
357         for i := range full {
358             n += copy(buf[n:], full[i])
359         }
360         copy(buf[n:], frag)
361         return buf, err
362     }
363
364     // ReadString reads until the first occurrence of delim in t
365     // returning a string containing the data up to and includin
366     // If ReadString encounters an error before finding a delimi
367     // it returns the data read before the error and the error i
368     // ReadString returns err != nil if and only if the returned
369     // delim.
370     func (b *Reader) ReadString(delim byte) (line string, err er
371         bytes, e := b.ReadBytes(delim)
372         return string(bytes), e
373     }
374
375     // buffered output
376
377     // Writer implements buffering for an io.Writer object.
378     // If an error occurs writing to a Writer, no more data will
379     // accepted and all subsequent writes will return the error.
380     type Writer struct {
381         err error
382         buf []byte
383         n    int
384         wr  io.Writer
385     }
386
387     // NewWriterSize returns a new Writer whose buffer has at le
388     // size. If the argument io.Writer is already a Writer with
389     // size, it returns the underlying Writer.
390     func NewWriterSize(wr io.Writer, size int) *Writer {

```

```

391         // Is it already a Writer?
392         b, ok := wr.(*Writer)
393         if ok && len(b.buf) >= size {
394             return b
395         }
396         if size <= 0 {
397             size = defaultBufSize
398         }
399         b = new(Writer)
400         b.buf = make([]byte, size)
401         b.wr = wr
402         return b
403     }
404
405     // NewWriter returns a new Writer whose buffer has the defau
406     func NewWriter(wr io.Writer) *Writer {
407         return NewWriterSize(wr, defaultBufSize)
408     }
409
410     // Flush writes any buffered data to the underlying io.Write
411     func (b *Writer) Flush() error {
412         if b.err != nil {
413             return b.err
414         }
415         if b.n == 0 {
416             return nil
417         }
418         n, e := b.wr.Write(b.buf[0:b.n])
419         if n < b.n && e == nil {
420             e = io.ErrShortWrite
421         }
422         if e != nil {
423             if n > 0 && n < b.n {
424                 copy(b.buf[0:b.n-n], b.buf[n:b.n])
425             }
426             b.n -= n
427             b.err = e
428             return e
429         }
430         b.n = 0
431         return nil
432     }
433
434     // Available returns how many bytes are unused in the buffer
435     func (b *Writer) Available() int { return len(b.buf) - b.n }
436
437     // Buffered returns the number of bytes that have been writt
438     func (b *Writer) Buffered() int { return b.n }
439

```

```

440 // Write writes the contents of p into the buffer.
441 // It returns the number of bytes written.
442 // If nn < len(p), it also returns an error explaining
443 // why the write is short.
444 func (b *Writer) Write(p []byte) (nn int, err error) {
445     for len(p) > b.Available() && b.err == nil {
446         var n int
447         if b.Buffered() == 0 {
448             // Large write, empty buffer.
449             // Write directly from p to avoid co
450             n, b.err = b.wr.Write(p)
451         } else {
452             n = copy(b.buf[b.n:], p)
453             b.n += n
454             b.Flush()
455         }
456         nn += n
457         p = p[n:]
458     }
459     if b.err != nil {
460         return nn, b.err
461     }
462     n := copy(b.buf[b.n:], p)
463     b.n += n
464     nn += n
465     return nn, nil
466 }
467
468 // WriteByte writes a single byte.
469 func (b *Writer) WriteByte(c byte) error {
470     if b.err != nil {
471         return b.err
472     }
473     if b.Available() <= 0 && b.Flush() != nil {
474         return b.err
475     }
476     b.buf[b.n] = c
477     b.n++
478     return nil
479 }
480
481 // WriteRune writes a single Unicode code point, returning
482 // the number of bytes written and any error.
483 func (b *Writer) WriteRune(r rune) (size int, err error) {
484     if r < utf8.RuneSelf {
485         err = b.WriteByte(byte(r))
486         if err != nil {
487             return 0, err
488         }

```

```

489         return 1, nil
490     }
491     if b.err != nil {
492         return 0, b.err
493     }
494     n := b.Available()
495     if n < utf8.UTFMax {
496         if b.Flush(); b.err != nil {
497             return 0, b.err
498         }
499         n = b.Available()
500         if n < utf8.UTFMax {
501             // Can only happen if buffer is still
502             return b.WriteString(string(r))
503         }
504     }
505     size = utf8.EncodeRune(b.buf[b.n:], r)
506     b.n += size
507     return size, nil
508 }
509
510 // WriteString writes a string.
511 // It returns the number of bytes written.
512 // If the count is less than len(s), it also returns an error
513 // why the write is short.
514 func (b *Writer) WriteString(s string) (int, error) {
515     nn := 0
516     for len(s) > b.Available() && b.err == nil {
517         n := copy(b.buf[b.n:], s)
518         b.n += n
519         nn += n
520         s = s[n:]
521         b.Flush()
522     }
523     if b.err != nil {
524         return nn, b.err
525     }
526     n := copy(b.buf[b.n:], s)
527     b.n += n
528     nn += n
529     return nn, nil
530 }
531
532 // buffered input and output
533
534 // ReadWriter stores pointers to a Reader and a Writer.
535 // It implements io.ReadWriter.
536 type ReadWriter struct {
537     *Reader
538     *Writer

```

```
539 }
540
541 // NewReadWriter allocates a new ReadWriter that dispatches
542 func NewReadWriter(r *Reader, w *Writer) *ReadWriter {
543     return &ReadWriter{r, w}
544 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/builtin/builtin.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6     Package builtin provides documentation for Go's prede
7     The items documented here are not actually in packag
8     but their descriptions here allow godoc to present d
9     for the language's special identifiers.
10 */
11 package builtin
12
13 // bool is the set of boolean values, true and false.
14 type bool bool
15
16 // uint8 is the set of all unsigned 8-bit integers.
17 // Range: 0 through 255.
18 type uint8 uint8
19
20 // uint16 is the set of all unsigned 16-bit integers.
21 // Range: 0 through 65535.
22 type uint16 uint16
23
24 // uint32 is the set of all unsigned 32-bit integers.
25 // Range: 0 through 4294967295.
26 type uint32 uint32
27
28 // uint64 is the set of all unsigned 64-bit integers.
29 // Range: 0 through 18446744073709551615.
30 type uint64 uint64
31
32 // int8 is the set of all signed 8-bit integers.
33 // Range: -128 through 127.
34 type int8 int8
35
36 // int16 is the set of all signed 16-bit integers.
37 // Range: -32768 through 32767.
38 type int16 int16
39
40 // int32 is the set of all signed 32-bit integers.
41 // Range: -2147483648 through 2147483647.
42 type int32 int32
43
44 // int64 is the set of all signed 64-bit integers.
```

```
45 // Range: -9223372036854775808 through 9223372036854775807.
46 type int64 int64
47
48 // float32 is the set of all IEEE-754 32-bit floating-point
49 type float32 float32
50
51 // float64 is the set of all IEEE-754 64-bit floating-point
52 type float64 float64
53
54 // complex64 is the set of all complex numbers with float32
55 // imaginary parts.
56 type complex64 complex64
57
58 // complex128 is the set of all complex numbers with float64
59 // imaginary parts.
60 type complex128 complex128
61
62 // string is the set of all strings of 8-bit bytes, conventi
63 // necessarily representing UTF-8-encoded text. A string may
64 // not nil. Values of string type are immutable.
65 type string string
66
67 // int is a signed integer type that is at least 32 bits in
68 // distinct type, however, and not an alias for, say, int32.
69 type int int
70
71 // uint is an unsigned integer type that is at least 32 bits
72 // distinct type, however, and not an alias for, say, uint32
73 type uint uint
74
75 // uintptr is an integer type that is large enough to hold t
76 // any pointer.
77 type uintptr uintptr
78
79 // byte is an alias for uint8 and is equivalent to uint8 in
80 // used, by convention, to distinguish byte values from 8-bi
81 // integer values.
82 type byte byte
83
84 // rune is an alias for int and is equivalent to int in all
85 // used, by convention, to distinguish character values from
86 // In a future version of Go, it will change to an alias of
87 type rune rune
88
89 // Type is here for the purposes of documentation only. It i
90 // for any Go type, but represents the same type for any giv
91 // invocation.
92 type Type int
93
94 // Type1 is here for the purposes of documentation only. It
```

```

95 // for any Go type, but represents the same type for any giv
96 // invocation.
97 type Type1 int
98
99 // IntegerType is here for the purposes of documentation onl
100 // for any integer type: int, uint, int8 etc.
101 type IntegerType int
102
103 // FloatType is here for the purposes of documentation only.
104 // for either float type: float32 or float64.
105 type FloatType float32
106
107 // ComplexType is here for the purposes of documentation onl
108 // stand-in for either complex type: complex64 or complex128
109 type ComplexType complex64
110
111 // The append built-in function appends elements to the end
112 // it has sufficient capacity, the destination is resliced t
113 // new elements. If it does not, a new underlying array will
114 // Append returns the updated slice. It is therefore necessa
115 // result of append, often in the variable holding the slice
116 //     slice = append(slice, elem1, elem2)
117 //     slice = append(slice, anotherSlice...)
118 func append(slice []Type, elems ...Type) []Type
119
120 // The copy built-in function copies elements from a source
121 // destination slice. (As a special case, it also will copy
122 // string to a slice of bytes.) The source and destination m
123 // returns the number of elements copied, which will be the
124 // len(src) and len(dst).
125 func copy(dst, src []Type) int
126
127 // The delete built-in function deletes the element with the
128 // (m[key]) from the map. If there is no such element, delet
129 // If m is nil, delete panics.
130 func delete(m map[Type]Type1, key Type)
131
132 // The len built-in function returns the length of v, accord
133 //     Array: the number of elements in v.
134 //     Pointer to array: the number of elements in *v (even
135 //     Slice, or map: the number of elements in v; if v is
136 //     String: the number of bytes in v.
137 //     Channel: the number of elements queued (unread) in t
138 //     if v is nil, len(v) is zero.
139 func len(v Type) int
140
141 // The cap built-in function returns the capacity of v, acco
142 //     Array: the number of elements in v (same as len(v)).
143 //     Pointer to array: the number of elements in *v (same

```

```

144 //      Slice: the maximum length the slice can reach when r
145 //      if v is nil, cap(v) is zero.
146 //      Channel: the channel buffer capacity, in units of el
147 //      if v is nil, cap(v) is zero.
148 func cap(v Type) int
149
150 // The make built-in function allocates and initializes an o
151 // slice, map, or chan (only). Like new, the first argument
152 // value. Unlike new, make's return type is the same as the
153 // argument, not a pointer to it. The specification of the r
154 // the type:
155 //      Slice: The size specifies the length. The capacity o
156 //      equal to its length. A second integer argument may b
157 //      specify a different capacity; it must be no smaller
158 //      length, so make([]int, 0, 10) allocates a slice of l
159 //      capacity 10.
160 //      Map: An initial allocation is made according to the
161 //      resulting map has length 0. The size may be omitted,
162 //      a small starting size is allocated.
163 //      Channel: The channel's buffer is initialized with th
164 //      buffer capacity. If zero, or the size is omitted, th
165 //      unbuffered.
166 func make(Type, size IntegerType) Type
167
168 // The new built-in function allocates memory. The first arg
169 // not a value, and the value returned is a pointer to a new
170 // allocated zero value of that type.
171 func new(Type) *Type
172
173 // The complex built-in function constructs a complex value
174 // floating-point values. The real and imaginary parts must
175 // size, either float32 or float64 (or assignable to them),
176 // value will be the corresponding complex type (complex64 f
177 // complex128 for float64).
178 func complex(r, i FloatType) ComplexType
179
180 // The real built-in function returns the real part of the c
181 // The return value will be floating point type correspondin
182 func real(c ComplexType) FloatType
183
184 // The imag built-in function returns the imaginary part of
185 // number c. The return value will be floating point type co
186 // the type of c.
187 func imag(c ComplexType) FloatType
188
189 // The close built-in function closes a channel, which must
190 // bidirectional or send-only. It should be executed only by
191 // never the receiver, and has the effect of shutting down t
192 // the last sent value is received. After the last value has

```

```

193 // from a closed channel c, any receive from c will succeed
194 // blocking, returning the zero value for the channel element
195 //      x, ok := <-c
196 // will also set ok to false for a closed channel.
197 func close(c chan<- Type)
198
199 // The panic built-in function stops normal execution of the
200 // goroutine. When a function F calls panic, normal execution
201 // immediately. Any functions whose execution was deferred by
202 // the usual way, and then F returns to its caller. To the caller's
203 // invocation of F then behaves like a call to panic, terminating
204 // execution and running any deferred functions. This continues until
205 // all functions in the executing goroutine have stopped, in reverse
206 // order. At that point, the program is terminated and the error condition
207 // is returned, including the value of the argument to panic. This termination
208 // is called panicking and can be controlled by the built-in function
209 // recover.
210 func panic(v interface{})
211
212 // The recover built-in function allows a program to manage a
213 // panicking goroutine. Executing a call to recover inside a
214 // function (but not any function called by it) stops the panicking
215 // goroutine by restoring normal execution and retrieves the error value
216 // returned by the call of panic. If recover is called outside the deferred
217 // goroutine, it does not stop a panicking sequence. In this case, or when the
218 // goroutine is not panicking, or if the argument supplied to panic was nil,
219 // recover returns nil. Thus the return value from recover reports whether the
220 // goroutine was panicking.
221 func recover() interface{}
222
223 // The error built-in interface type is the conventional interface type
224 // representing an error condition, with the nil value representing no
225 // error.
226 type error interface {
227     Error() string

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/bytes/buffer.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package bytes
6
7 // Simple byte buffer for marshaling data.
8
9 import (
10     "errors"
11     "io"
12     "unicode/utf8"
13 )
14
15 // A Buffer is a variable-sized buffer of bytes with Read and
16 // Write methods. The zero value for Buffer is an empty buffer ready to use.
17 type Buffer struct {
18     buf      []byte // contents are the byte slice
19     off      int    // read at &buf[off], write at &buf[off+len(buf)-len(buf)+off]
20     runeBytes [utf8.UTFMax]byte // avoid allocation of slice
21     bootstrap [64]byte // memory to hold first read operation,
22     lastRead  readOp // last read operation,
23 }
24
25 // The readOp constants describe the last action performed on
26 // the buffer, so that UnreadRune and UnreadByte can
27 // check for invalid usage.
28 type readOp int
29
30 const (
31     opInvalid readOp = iota // Non-read operation.
32     opReadRune              // Read rune.
33     opRead                  // Any other read operation
34 )
35
36 // ErrTooLarge is passed to panic if memory cannot be allocated.
37 var ErrTooLarge = errors.New("bytes.Buffer: too large")
38
39 // Bytes returns a slice of the contents of the unread portion of
40 // the buffer. If the caller changes the slice, the contents of the
41 // buffer will change. If there are no intervening method calls on the
42 // Buffer, the slice is valid until the next call to Bytes.
43 func (b *Buffer) Bytes() []byte { return b.buf[b.off:] }
44
```

```

45 // String returns the contents of the unread portion of the
46 // as a string. If the Buffer is a nil pointer, it returns
47 func (b *Buffer) String() string {
48     if b == nil {
49         // Special case, useful in debugging.
50         return "<nil>"
51     }
52     return string(b.buf[b.off:])
53 }
54
55 // Len returns the number of bytes of the unread portion of
56 // b.Len() == len(b.Bytes()).
57 func (b *Buffer) Len() int { return len(b.buf) - b.off }
58
59 // Truncate discards all but the first n unread bytes from t
60 // It panics if n is negative or greater than the length of
61 func (b *Buffer) Truncate(n int) {
62     b.lastRead = opInvalid
63     switch {
64     case n < 0 || n > b.Len():
65         panic("bytes.Buffer: truncation out of range")
66     case n == 0:
67         // Reuse buffer space.
68         b.off = 0
69     }
70     b.buf = b.buf[0 : b.off+n]
71 }
72
73 // Reset resets the buffer so it has no content.
74 // b.Reset() is the same as b.Truncate(0).
75 func (b *Buffer) Reset() { b.Truncate(0) }
76
77 // grow grows the buffer to guarantee space for n more bytes
78 // It returns the index where bytes should be written.
79 // If the buffer can't grow it will panic with ErrTooLarge.
80 func (b *Buffer) grow(n int) int {
81     m := b.Len()
82     // If buffer is empty, reset to recover space.
83     if m == 0 && b.off != 0 {
84         b.Truncate(0)
85     }
86     if len(b.buf)+n > cap(b.buf) {
87         var buf []byte
88         if b.buf == nil && n <= len(b.bootstrap) {
89             buf = b.bootstrap[0:]
90         } else {
91             // not enough space anywhere
92             buf = makeSlice(2*cap(b.buf) + n)
93             copy(buf, b.buf[b.off:])
94         }

```

```

95             b.buf = buf
96             b.off = 0
97         }
98         b.buf = b.buf[0 : b.off+m+n]
99         return b.off + m
100    }
101
102    // Write appends the contents of p to the buffer. The return
103    // value n is the length of p; err is always nil.
104    // If the buffer becomes too large, Write will panic with
105    // ErrTooLarge.
106    func (b *Buffer) Write(p []byte) (n int, err error) {
107        b.lastRead = opInvalid
108        m := b.grow(len(p))
109        return copy(b.buf[m:], p), nil
110    }
111
112    // WriteString appends the contents of s to the buffer. The
113    // value n is the length of s; err is always nil.
114    // If the buffer becomes too large, WriteString will panic with
115    // ErrTooLarge.
116    func (b *Buffer) WriteString(s string) (n int, err error) {
117        b.lastRead = opInvalid
118        m := b.grow(len(s))
119        return copy(b.buf[m:], s), nil
120    }
121
122    // MinRead is the minimum slice size passed to a Read call b
123    // Buffer.ReadFrom. As long as the Buffer has at least MinR
124    // what is required to hold the contents of r, ReadFrom will
125    // underlying buffer.
126    const MinRead = 512
127
128    // ReadFrom reads data from r until EOF and appends it to th
129    // The return value n is the number of bytes read.
130    // Any error except io.EOF encountered during the read
131    // is also returned.
132    // If the buffer becomes too large, ReadFrom will panic with
133    // ErrTooLarge.
134    func (b *Buffer) ReadFrom(r io.Reader) (n int64, err error)
135        b.lastRead = opInvalid
136        // If buffer is empty, reset to recover space.
137        if b.off >= len(b.buf) {
138            b.Truncate(0)
139        }
140        for {
141            if free := cap(b.buf) - len(b.buf); free < M
142                // not enough space at end
143                newBuf := b.buf

```

```

144         if b.off+free < MinRead {
145             // not enough space using be
146             // double buffer capacity
147             newBuf = makeSlice(2*cap(b.b
148         })
149         copy(newBuf, b.buf[b.off:])
150         b.buf = newBuf[:len(b.buf)-b.off]
151         b.off = 0
152     }
153     m, e := r.Read(b.buf[len(b.buf):cap(b.buf)])
154     b.buf = b.buf[0 : len(b.buf)+m]
155     n += int64(m)
156     if e == io.EOF {
157         break
158     }
159     if e != nil {
160         return n, e
161     }
162 }
163 return n, nil // err is EOF, so return nil explicitl
164 }
165
166 // makeSlice allocates a slice of size n. If the allocation
167 // with ErrTooLarge.
168 func makeSlice(n int) []byte {
169     // If the make fails, give a known error.
170     defer func() {
171         if recover() != nil {
172             panic(ErrTooLarge)
173         }
174     }()
175     return make([]byte, n)
176 }
177
178 // WriteTo writes data to w until the buffer is drained or a
179 // occurs. The return value n is the number of bytes written
180 // fits into an int, but it is int64 to match the io.WriterT
181 // Any error encountered during the write is also returned.
182 func (b *Buffer) WriteTo(w io.Writer) (n int64, err error) {
183     b.lastRead = opInvalid
184     if b.off < len(b.buf) {
185         nBytes := b.Len()
186         m, e := w.Write(b.buf[b.off:])
187         if m > nBytes {
188             panic("bytes.Buffer.WriteTo: invalid
189         }
190         b.off += m
191         n = int64(m)
192         if e != nil {

```

```

193             return n, e
194         }
195         // all bytes should have been written, by de
196         // Write method in io.Writer
197         if m != nBytes {
198             return n, io.ErrShortWrite
199         }
200     }
201     // Buffer is now empty; reset.
202     b.Truncate(0)
203     return
204 }
205
206 // WriteByte appends the byte c to the buffer.
207 // The returned error is always nil, but is included
208 // to match bufio.Writer's WriteByte.
209 // If the buffer becomes too large, WriteByte will panic wit
210 // ErrTooLarge.
211 func (b *Buffer) WriteByte(c byte) error {
212     b.lastRead = opInvalid
213     m := b.grow(1)
214     b.buf[m] = c
215     return nil
216 }
217
218 // WriteRune appends the UTF-8 encoding of Unicode
219 // code point r to the buffer, returning its length and
220 // an error, which is always nil but is included
221 // to match bufio.Writer's WriteRune.
222 // If the buffer becomes too large, WriteRune will panic wit
223 // ErrTooLarge.
224 func (b *Buffer) WriteRune(r rune) (n int, err error) {
225     if r < utf8.RuneSelf {
226         b.WriteByte(byte(r))
227         return 1, nil
228     }
229     n = utf8.EncodeRune(b.runeBytes[0:], r)
230     b.Write(b.runeBytes[0:n])
231     return n, nil
232 }
233
234 // Read reads the next len(p) bytes from the buffer or until
235 // is drained. The return value n is the number of bytes re
236 // buffer has no data to return, err is io.EOF (unless len(p)
237 // otherwise it is nil.
238 func (b *Buffer) Read(p []byte) (n int, err error) {
239     b.lastRead = opInvalid
240     if b.off >= len(b.buf) {
241         // Buffer is empty, reset to recover space.
242         b.Truncate(0)

```

```

243             if len(p) == 0 {
244                 return
245             }
246             return 0, io.EOF
247         }
248         n = copy(p, b.buf[b.off:])
249         b.off += n
250         if n > 0 {
251             b.lastRead = opRead
252         }
253         return
254     }
255
256     // Next returns a slice containing the next n bytes from the
257     // advancing the buffer as if the bytes had been returned by
258     // If there are fewer than n bytes in the buffer, Next return
259     // The slice is only valid until the next call to a read or
260     func (b *Buffer) Next(n int) []byte {
261         b.lastRead = opInvalid
262         m := b.Len()
263         if n > m {
264             n = m
265         }
266         data := b.buf[b.off : b.off+n]
267         b.off += n
268         if n > 0 {
269             b.lastRead = opRead
270         }
271         return data
272     }
273
274     // ReadByte reads and returns the next byte from the buffer.
275     // If no byte is available, it returns error io.EOF.
276     func (b *Buffer) ReadByte() (c byte, err error) {
277         b.lastRead = opInvalid
278         if b.off >= len(b.buf) {
279             // Buffer is empty, reset to recover space.
280             b.Truncate(0)
281             return 0, io.EOF
282         }
283         c = b.buf[b.off]
284         b.off++
285         b.lastRead = opRead
286         return c, nil
287     }
288
289     // ReadRune reads and returns the next UTF-8-encoded
290     // Unicode code point from the buffer.
291     // If no bytes are available, the error returned is io.EOF.

```

```

292 // If the bytes are an erroneous UTF-8 encoding, it
293 // consumes one byte and returns U+FFFD, 1.
294 func (b *Buffer) ReadRune() (r rune, size int, err error) {
295     b.lastRead = opInvalid
296     if b.off >= len(b.buf) {
297         // Buffer is empty, reset to recover space.
298         b.Truncate(0)
299         return 0, 0, io.EOF
300     }
301     b.lastRead = opReadRune
302     c := b.buf[b.off]
303     if c < utf8.RuneSelf {
304         b.off++
305         return rune(c), 1, nil
306     }
307     r, n := utf8.DecodeRune(b.buf[b.off:])
308     b.off += n
309     return r, n, nil
310 }
311
312 // UnreadRune unreads the last rune returned by ReadRune.
313 // If the most recent read or write operation on the buffer
314 // not a ReadRune, UnreadRune returns an error. (In this re
315 // it is stricter than UnreadByte, which will unread the las
316 // from any read operation.)
317 func (b *Buffer) UnreadRune() error {
318     if b.lastRead != opReadRune {
319         return errors.New("bytes.Buffer: UnreadRune:
320     }
321     b.lastRead = opInvalid
322     if b.off > 0 {
323         _, n := utf8.DecodeLastRune(b.buf[0:b.off])
324         b.off -= n
325     }
326     return nil
327 }
328
329 // UnreadByte unreads the last byte returned by the most rec
330 // read operation. If write has happened since the last rea
331 // returns an error.
332 func (b *Buffer) UnreadByte() error {
333     if b.lastRead != opReadRune && b.lastRead != opRead
334         return errors.New("bytes.Buffer: UnreadByte:
335     }
336     b.lastRead = opInvalid
337     if b.off > 0 {
338         b.off--
339     }
340     return nil

```

```

341 }
342
343 // ReadBytes reads until the first occurrence of delim in th
344 // returning a slice containing the data up to and including
345 // If ReadBytes encounters an error before finding a delimit
346 // it returns the data read before the error and the error i
347 // ReadBytes returns err != nil if and only if the returned
348 // delim.
349 func (b *Buffer) ReadBytes(delim byte) (line []byte, err error) {
350     i := IndexByte(b.buf[b.off:], delim)
351     size := i + 1
352     if i < 0 {
353         size = len(b.buf) - b.off
354         err = io.EOF
355     }
356     line = make([]byte, size)
357     copy(line, b.buf[b.off:])
358     b.off += size
359     return
360 }
361
362 // ReadString reads until the first occurrence of delim in t
363 // returning a string containing the data up to and includin
364 // If ReadString encounters an error before finding a delimi
365 // it returns the data read before the error and the error i
366 // ReadString returns err != nil if and only if the returned
367 // in delim.
368 func (b *Buffer) ReadString(delim byte) (line string, err error) {
369     bytes, err := b.ReadBytes(delim)
370     return string(bytes), err
371 }
372
373 // NewBuffer creates and initializes a new Buffer using buf
374 // contents. It is intended to prepare a Buffer to read exi
375 // can also be used to size the internal buffer for writing.
376 // buf should have the desired capacity but a length of zero
377 //
378 // In most cases, new(Buffer) (or just declaring a Buffer va
379 // sufficient to initialize a Buffer.
380 func NewBuffer(buf []byte) *Buffer { return &Buffer{buf: buf
381
382 // NewBufferString creates and initializes a new Buffer usin
383 // initial contents. It is intended to prepare a buffer to r
384 // string.
385 //
386 // In most cases, new(Buffer) (or just declaring a Buffer va
387 // sufficient to initialize a Buffer.
388 func NewBufferString(s string) *Buffer {
389     return &Buffer{buf: []byte(s)}
390 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/bytes/bytes.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package bytes implements functions for the manipulation o
6 // It is analogous to the facilities of the strings package.
7 package bytes
8
9 import (
10     "unicode"
11     "unicode/utf8"
12 )
13
14 // Compare returns an integer comparing the two byte arrays
15 // The result will be 0 if a==b, -1 if a < b, and +1 if a >
16 // A nil argument is equivalent to an empty slice.
17 func Compare(a, b []byte) int {
18     m := len(a)
19     if m > len(b) {
20         m = len(b)
21     }
22     for i, ac := range a[0:m] {
23         bc := b[i]
24         switch {
25             case ac > bc:
26                 return 1
27             case ac < bc:
28                 return -1
29         }
30     }
31     switch {
32     case len(a) < len(b):
33         return -1
34     case len(a) > len(b):
35         return 1
36     }
37     return 0
38 }
39
40 // Equal returns a boolean reporting whether a == b.
41 // A nil argument is equivalent to an empty slice.
42 func Equal(a, b []byte) bool
43
44 func equalPortable(a, b []byte) bool {
```

```

45     if len(a) != len(b) {
46         return false
47     }
48     for i, c := range a {
49         if c != b[i] {
50             return false
51         }
52     }
53     return true
54 }
55
56 // explode splits s into an array of UTF-8 sequences, one pe
57 // up to a maximum of n byte arrays. Invalid UTF-8 sequences
58 func explode(s []byte, n int) [][]byte {
59     if n <= 0 {
60         n = len(s)
61     }
62     a := make([][]byte, n)
63     var size int
64     na := 0
65     for len(s) > 0 {
66         if na+1 >= n {
67             a[na] = s
68             na++
69             break
70         }
71         _, size = utf8.DecodeRune(s)
72         a[na] = s[0:size]
73         s = s[size:]
74         na++
75     }
76     return a[0:na]
77 }
78
79 // Count counts the number of non-overlapping instances of s
80 func Count(s, sep []byte) int {
81     n := len(sep)
82     if n == 0 {
83         return utf8.RuneCount(s) + 1
84     }
85     if n > len(s) {
86         return 0
87     }
88     count := 0
89     c := sep[0]
90     i := 0
91     t := s[:len(s)-n+1]
92     for i < len(t) {
93         if t[i] != c {
94             o := IndexByte(t[i:], c)

```

```

95             if o < 0 {
96                 break
97             }
98             i += o
99         }
100         if n == 1 || Equal(s[i:i+n], sep) {
101             count++
102             i += n
103             continue
104         }
105         i++
106     }
107     return count
108 }
109
110 // Contains returns whether subslice is within b.
111 func Contains(b, subslice []byte) bool {
112     return Index(b, subslice) != -1
113 }
114
115 // Index returns the index of the first instance of sep in s
116 func Index(s, sep []byte) int {
117     n := len(sep)
118     if n == 0 {
119         return 0
120     }
121     if n > len(s) {
122         return -1
123     }
124     c := sep[0]
125     if n == 1 {
126         return IndexByte(s, c)
127     }
128     i := 0
129     t := s[:len(s)-n+1]
130     for i < len(t) {
131         if t[i] != c {
132             o := IndexByte(t[i:], c)
133             if o < 0 {
134                 break
135             }
136             i += o
137         }
138         if Equal(s[i:i+n], sep) {
139             return i
140         }
141         i++
142     }
143     return -1

```

```

144 }
145
146 func indexBytePortable(s []byte, c byte) int {
147     for i, b := range s {
148         if b == c {
149             return i
150         }
151     }
152     return -1
153 }
154
155 // LastIndex returns the index of the last instance of sep i
156 func LastIndex(s, sep []byte) int {
157     n := len(sep)
158     if n == 0 {
159         return len(s)
160     }
161     c := sep[0]
162     for i := len(s) - n; i >= 0; i-- {
163         if s[i] == c && (n == 1 || Equal(s[i:i+n], s
164             return i
165         }
166     }
167     return -1
168 }
169
170 // IndexRune interprets s as a sequence of UTF-8-encoded Uni
171 // It returns the byte index of the first occurrence in s of
172 // It returns -1 if rune is not present in s.
173 func IndexRune(s []byte, r rune) int {
174     for i := 0; i < len(s); {
175         r1, size := utf8.DecodeRune(s[i:])
176         if r == r1 {
177             return i
178         }
179         i += size
180     }
181     return -1
182 }
183
184 // IndexAny interprets s as a sequence of UTF-8-encoded Unic
185 // It returns the byte index of the first occurrence in s of
186 // code points in chars. It returns -1 if chars is empty or
187 // point in common.
188 func IndexAny(s []byte, chars string) int {
189     if len(chars) > 0 {
190         var r rune
191         var width int
192         for i := 0; i < len(s); i += width {

```

```

193         r = rune(s[i])
194         if r < utf8.RuneSelf {
195             width = 1
196         } else {
197             r, width = utf8.DecodeRune(s
198         }
199         for _, ch := range chars {
200             if r == ch {
201                 return i
202             }
203         }
204     }
205 }
206 return -1
207 }
208
209 // LastIndexAny interprets s as a sequence of UTF-8-encoded
210 // points. It returns the byte index of the last occurrence
211 // the Unicode code points in chars. It returns -1 if chars
212 // there is no code point in common.
213 func LastIndexAny(s []byte, chars string) int {
214     if len(chars) > 0 {
215         for i := len(s); i > 0; {
216             r, size := utf8.DecodeLastRune(s[0:i]
217             i -= size
218             for _, ch := range chars {
219                 if r == ch {
220                     return i
221                 }
222             }
223         }
224     }
225     return -1
226 }
227
228 // Generic split: splits after each instance of sep,
229 // including sepSave bytes of sep in the subarrays.
230 func genSplit(s, sep []byte, sepSave, n int) [][]byte {
231     if n == 0 {
232         return nil
233     }
234     if len(sep) == 0 {
235         return explode(s, n)
236     }
237     if n < 0 {
238         n = Count(s, sep) + 1
239     }
240     c := sep[0]
241     start := 0
242     a := make([][]byte, n)

```

```

243         na := 0
244         for i := 0; i+len(sep) <= len(s) && na+1 < n; i++ {
245             if s[i] == c && (len(sep) == 1 || Equal(s[i:
246                 a[na] = s[start : i+sepSave]
247                 na++
248                 start = i + len(sep)
249                 i += len(sep) - 1
250             }
251         }
252         a[na] = s[start:]
253         return a[0 : na+1]
254     }
255
256 // SplitN slices s into subslices separated by sep and retur
257 // the subslices between those separators.
258 // If sep is empty, SplitN splits after each UTF-8 sequence.
259 // The count determines the number of subslices to return:
260 //   n > 0: at most n subslices; the last subslice will be t
261 //   n == 0: the result is nil (zero subslices)
262 //   n < 0: all subslices
263 func SplitN(s, sep []byte, n int) [][]byte { return genSplit
264
265 // SplitAfterN slices s into subslices after each instance o
266 // returns a slice of those subslices.
267 // If sep is empty, SplitAfterN splits after each UTF-8 sequ
268 // The count determines the number of subslices to return:
269 //   n > 0: at most n subslices; the last subslice will be t
270 //   n == 0: the result is nil (zero subslices)
271 //   n < 0: all subslices
272 func SplitAfterN(s, sep []byte, n int) [][]byte {
273     return genSplit(s, sep, len(sep), n)
274 }
275
276 // Split slices s into all subslices separated by sep and re
277 // the subslices between those separators.
278 // If sep is empty, Split splits after each UTF-8 sequence.
279 // It is equivalent to SplitN with a count of -1.
280 func Split(s, sep []byte) [][]byte { return genSplit(s, sep,
281
282 // SplitAfter slices s into all subslices after each instanc
283 // returns a slice of those subslices.
284 // If sep is empty, SplitAfter splits after each UTF-8 seque
285 // It is equivalent to SplitAfterN with a count of -1.
286 func SplitAfter(s, sep []byte) [][]byte {
287     return genSplit(s, sep, len(sep), -1)
288 }
289
290 // Fields splits the array s around each instance of one or
291 // characters, returning a slice of subarrays of s or an emp

```

```

292 func Fields(s []byte) [][]byte {
293     return FieldsFunc(s, unicode.IsSpace)
294 }
295
296 // FieldsFunc interprets s as a sequence of UTF-8-encoded Un
297 // It splits the array s at each run of code points c satisf
298 // returns a slice of subarrays of s. If no code points in
299 // empty slice is returned.
300 func FieldsFunc(s []byte, f func(rune) bool) [][]byte {
301     n := 0
302     inField := false
303     for i := 0; i < len(s); {
304         r, size := utf8.DecodeRune(s[i:])
305         wasInField := inField
306         inField = !f(r)
307         if inField && !wasInField {
308             n++
309         }
310         i += size
311     }
312
313     a := make([][]byte, n)
314     na := 0
315     fieldStart := -1
316     for i := 0; i <= len(s) && na < n; {
317         r, size := utf8.DecodeRune(s[i:])
318         if fieldStart < 0 && size > 0 && !f(r) {
319             fieldStart = i
320             i += size
321             continue
322         }
323         if fieldStart >= 0 && (size == 0 || f(r)) {
324             a[na] = s[fieldStart:i]
325             na++
326             fieldStart = -1
327         }
328         if size == 0 {
329             break
330         }
331         i += size
332     }
333     return a[0:na]
334 }
335
336 // Join concatenates the elements of a to create a single by
337 // sep is placed between elements in the resulting array.
338 func Join(a [][]byte, sep []byte) []byte {
339     if len(a) == 0 {
340         return []byte{}

```

```

341     }
342     if len(a) == 1 {
343         return a[0]
344     }
345     n := len(sep) * (len(a) - 1)
346     for i := 0; i < len(a); i++ {
347         n += len(a[i])
348     }
349
350     b := make([]byte, n)
351     bp := copy(b, a[0])
352     for _, s := range a[1:] {
353         bp += copy(b[bp:], sep)
354         bp += copy(b[bp:], s)
355     }
356     return b
357 }
358
359 // HasPrefix tests whether the byte array s begins with pref
360 func HasPrefix(s, prefix []byte) bool {
361     return len(s) >= len(prefix) && Equal(s[0:len(prefix)
362 }
363
364 // HasSuffix tests whether the byte array s ends with suffix
365 func HasSuffix(s, suffix []byte) bool {
366     return len(s) >= len(suffix) && Equal(s[len(s)-len(s
367 }
368
369 // Map returns a copy of the byte array s with all its chara
370 // according to the mapping function. If mapping returns a n
371 // dropped from the string with no replacement. The charact
372 // output are interpreted as UTF-8-encoded Unicode code poin
373 func Map(mapping func(r rune) rune, s []byte) []byte {
374     // In the worst case, the array can grow when mapped
375     // things unpleasant. But it's so rare we barge in
376     // fine. It could also shrink but that falls out na
377     maxbytes := len(s) // length of b
378     nbytes := 0        // number of bytes encoded in b
379     b := make([]byte, maxbytes)
380     for i := 0; i < len(s); {
381         wid := 1
382         r := rune(s[i])
383         if r >= utf8.RuneSelf {
384             r, wid = utf8.DecodeRune(s[i:])
385         }
386         r = mapping(r)
387         if r >= 0 {
388             if nbytes+utf8.RuneLen(r) > maxbytes
389                 // Grow the buffer.
390                 maxbytes = maxbytes*2 + utf8

```

```

391             nb := make([]byte, maxbytes)
392             copy(nb, b[0:nbytes])
393             b = nb
394         }
395         nbytes += utf8.EncodeRune(b[nbytes:r]
396     }
397     i += wid
398 }
399 return b[0:nbytes]
400 }
401
402 // Repeat returns a new byte slice consisting of count copie
403 func Repeat(b []byte, count int) []byte {
404     nb := make([]byte, len(b)*count)
405     bp := 0
406     for i := 0; i < count; i++ {
407         for j := 0; j < len(b); j++ {
408             nb[bp] = b[j]
409             bp++
410         }
411     }
412     return nb
413 }
414
415 // ToUpper returns a copy of the byte array s with all Unico
416 func ToUpper(s []byte) []byte { return Map(unicode.ToUpper,
417
418 // ToUpper returns a copy of the byte array s with all Unico
419 func ToLower(s []byte) []byte { return Map(unicode.ToLower,
420
421 // ToTitle returns a copy of the byte array s with all Unico
422 func ToTitle(s []byte) []byte { return Map(unicode.ToTitle,
423
424 // ToUpperSpecial returns a copy of the byte array s with al
425 // upper case, giving priority to the special casing rules.
426 func ToUpperSpecial(_case unicode.SpecialCase, s []byte) []b
427     return Map(func(r rune) rune { return _case.ToUpper(
428 }
429
430 // ToLowerSpecial returns a copy of the byte array s with al
431 // lower case, giving priority to the special casing rules.
432 func ToLowerSpecial(_case unicode.SpecialCase, s []byte) []b
433     return Map(func(r rune) rune { return _case.ToLower(
434 }
435
436 // ToTitleSpecial returns a copy of the byte array s with al
437 // title case, giving priority to the special casing rules.
438 func ToTitleSpecial(_case unicode.SpecialCase, s []byte) []b
439     return Map(func(r rune) rune { return _case.ToTitle(

```

```

440 }
441
442 // isSeparator reports whether the rune could mark a word bo
443 // TODO: update when package unicode captures more of the pr
444 func isSeparator(r rune) bool {
445     // ASCII alphanumerics and underscore are not separa
446     if r <= 0x7F {
447         switch {
448             case '0' <= r && r <= '9':
449                 return false
450             case 'a' <= r && r <= 'z':
451                 return false
452             case 'A' <= r && r <= 'Z':
453                 return false
454             case r == '_':
455                 return false
456         }
457         return true
458     }
459     // Letters and digits are not separators
460     if unicode.IsLetter(r) || unicode.IsDigit(r) {
461         return false
462     }
463     // Otherwise, all we can do for now is treat spaces
464     return unicode.IsSpace(r)
465 }
466
467 // BUG(r): The rule Title uses for word boundaries does not
468
469 // Title returns a copy of s with all Unicode letters that b
470 // mapped to their title case.
471 func Title(s []byte) []byte {
472     // Use a closure here to remember state.
473     // Hackish but effective. Depends on Map scanning in
474     // the closure once per rune.
475     prev := ' '
476     return Map(
477         func(r rune) rune {
478             if isSeparator(prev) {
479                 prev = r
480                 return unicode.ToTitle(r)
481             }
482             prev = r
483             return r
484         },
485         s)
486 }
487
488 // TrimLeftFunc returns a subslice of s by slicing off all l

```

```

489 // Unicode code points c that satisfy f(c).
490 func TrimLeftFunc(s []byte, f func(r rune) bool) []byte {
491     i := indexFunc(s, f, false)
492     if i == -1 {
493         return nil
494     }
495     return s[i:]
496 }
497
498 // TrimRightFunc returns a subslice of s by slicing off all
499 // encoded Unicode code points c that satisfy f(c).
500 func TrimRightFunc(s []byte, f func(r rune) bool) []byte {
501     i := lastIndexFunc(s, f, false)
502     if i >= 0 && s[i] >= utf8.RuneSelf {
503         _, wid := utf8.DecodeRune(s[i:])
504         i += wid
505     } else {
506         i++
507     }
508     return s[0:i]
509 }
510
511 // TrimFunc returns a subslice of s by slicing off all leadin
512 // UTF-8-encoded Unicode code points c that satisfy f(c).
513 func TrimFunc(s []byte, f func(r rune) bool) []byte {
514     return TrimRightFunc(TrimLeftFunc(s, f), f)
515 }
516
517 // IndexFunc interprets s as a sequence of UTF-8-encoded Uni
518 // It returns the byte index in s of the first Unicode
519 // code point satisfying f(c), or -1 if none do.
520 func IndexFunc(s []byte, f func(r rune) bool) int {
521     return indexFunc(s, f, true)
522 }
523
524 // LastIndexFunc interprets s as a sequence of UTF-8-encoded
525 // It returns the byte index in s of the last Unicode
526 // code point satisfying f(c), or -1 if none do.
527 func LastIndexFunc(s []byte, f func(r rune) bool) int {
528     return lastIndexFunc(s, f, true)
529 }
530
531 // indexFunc is the same as IndexFunc except that if
532 // truth==false, the sense of the predicate function is
533 // inverted.
534 func indexFunc(s []byte, f func(r rune) bool, truth bool) int {
535     start := 0
536     for start < len(s) {
537         wid := 1
538         r := rune(s[start])

```

```

539         if r >= utf8.RuneSelf {
540             r, wid = utf8.DecodeRune(s[start:])
541         }
542         if f(r) == truth {
543             return start
544         }
545         start += wid
546     }
547     return -1
548 }
549
550 // lastIndexFunc is the same as LastIndexFunc except that if
551 // truth==false, the sense of the predicate function is
552 // inverted.
553 func lastIndexFunc(s []byte, f func(r rune) bool, truth bool)
554     for i := len(s); i > 0; {
555         r, size := utf8.DecodeLastRune(s[0:i])
556         i -= size
557         if f(r) == truth {
558             return i
559         }
560     }
561     return -1
562 }
563
564 func makeCutsetFunc(cutset string) func(r rune) bool {
565     return func(r rune) bool {
566         for _, c := range cutset {
567             if c == r {
568                 return true
569             }
570         }
571         return false
572     }
573 }
574
575 // Trim returns a subslice of s by slicing off all leading a
576 // trailing UTF-8-encoded Unicode code points contained in c
577 func Trim(s []byte, cutset string) []byte {
578     return TrimFunc(s, makeCutsetFunc(cutset))
579 }
580
581 // TrimLeft returns a subslice of s by slicing off all leadi
582 // UTF-8-encoded Unicode code points contained in cutset.
583 func TrimLeft(s []byte, cutset string) []byte {
584     return TrimLeftFunc(s, makeCutsetFunc(cutset))
585 }
586
587 // TrimRight returns a subslice of s by slicing off all trai

```

```

588 // UTF-8-encoded Unicode code points that are contained in c
589 func TrimRight(s []byte, cutset string) []byte {
590     return TrimRightFunc(s, makeCutsetFunc(cutset))
591 }
592
593 // TrimSpace returns a subslice of s by slicing off all lead
594 // trailing white space, as defined by Unicode.
595 func TrimSpace(s []byte) []byte {
596     return TrimFunc(s, unicode.IsSpace)
597 }
598
599 // Runes returns a slice of runes (Unicode code points) equi
600 func Runes(s []byte) []rune {
601     t := make([]rune, utf8.RuneCount(s))
602     i := 0
603     for len(s) > 0 {
604         r, l := utf8.DecodeRune(s)
605         t[i] = r
606         i++
607         s = s[l:]
608     }
609     return t
610 }
611
612 // Replace returns a copy of the slice s with the first n
613 // non-overlapping instances of old replaced by new.
614 // If n < 0, there is no limit on the number of replacements
615 func Replace(s, old, new []byte, n int) []byte {
616     m := 0
617     if n != 0 {
618         // Compute number of replacements.
619         m = Count(s, old)
620     }
621     if m == 0 {
622         // Nothing to do. Just copy.
623         t := make([]byte, len(s))
624         copy(t, s)
625         return t
626     }
627     if n < 0 || m < n {
628         n = m
629     }
630
631     // Apply replacements to buffer.
632     t := make([]byte, len(s)+n*(len(new)-len(old)))
633     w := 0
634     start := 0
635     for i := 0; i < n; i++ {
636         j := start

```

```

637         if len(old) == 0 {
638             if i > 0 {
639                 _, wid := utf8.DecodeRune(s[
640                     j += wid
641                 ])
642             } else {
643                 j += Index(s[start:], old)
644             }
645             w += copy(t[w:], s[start:j])
646             w += copy(t[w:], new)
647             start = j + len(old)
648         }
649         w += copy(t[w:], s[start:])
650         return t[0:w]
651     }
652
653     // EqualFold reports whether s and t, interpreted as UTF-8 s
654     // are equal under Unicode case-folding.
655     func EqualFold(s, t []byte) bool {
656         for len(s) != 0 && len(t) != 0 {
657             // Extract first rune from each.
658             var sr, tr rune
659             if s[0] < utf8.RuneSelf {
660                 sr, s = rune(s[0]), s[1:]
661             } else {
662                 r, size := utf8.DecodeRune(s)
663                 sr, s = r, s[size:]
664             }
665             if t[0] < utf8.RuneSelf {
666                 tr, t = rune(t[0]), t[1:]
667             } else {
668                 r, size := utf8.DecodeRune(t)
669                 tr, t = r, t[size:]
670             }
671
672             // If they match, keep going; if not, return
673
674             // Easy case.
675             if tr == sr {
676                 continue
677             }
678
679             // Make sr < tr to simplify what follows.
680             if tr < sr {
681                 tr, sr = sr, tr
682             }
683             // Fast check for ASCII.
684             if tr < utf8.RuneSelf && 'A' <= sr && sr <=
685                 // ASCII, and sr is upper case. tr
686                 if tr == sr+'a'-'A' {

```

```

687             continue
688         }
689         return false
690     }
691
692     // General case. SimpleFold(x) returns the
693     // or wraps around to smaller values.
694     r := unicode.SimpleFold(sr)
695     for r != sr && r < tr {
696         r = unicode.SimpleFold(r)
697     }
698     if r == tr {
699         continue
700     }
701     return false
702 }
703
704 // One string is empty. Are both?
705 return len(s) == len(t)
706 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/bytes/bytes_decl.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.  
2 // Use of this source code is governed by a BSD-style  
3 // license that can be found in the LICENSE file.  
4  
5 package bytes  
6  
7 // IndexByte returns the index of the first instance of c in  
8 func IndexByte(s []byte, c byte) int // asm_${GOARCH}.s
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/bytes/reader.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package bytes
6
7 import (
8     "errors"
9     "io"
10    "unicode/utf8"
11 )
12
13 // A Reader implements the io.Reader, io.ReaderAt, io.Seeker
14 // io.ByteScanner, and io.RuneScanner interfaces by reading
15 // a byte slice.
16 // Unlike a Buffer, a Reader is read-only and supports seeki
17 type Reader struct {
18     s      []byte
19     i      int // current reading index
20     prevRune int // index of previous rune; or < 0
21 }
22
23 // Len returns the number of bytes of the unread portion of
24 // slice.
25 func (r *Reader) Len() int {
26     if r.i >= len(r.s) {
27         return 0
28     }
29     return len(r.s) - r.i
30 }
31
32 func (r *Reader) Read(b []byte) (n int, err error) {
33     if len(b) == 0 {
34         return 0, nil
35     }
36     if r.i >= len(r.s) {
37         return 0, io.EOF
38     }
39     n = copy(b, r.s[r.i:])
40     r.i += n
41     r.prevRune = -1
42     return
43 }
44
```

```

45 func (r *Reader) ReadAt(b []byte, off int64) (n int, err error) {
46     if off < 0 {
47         return 0, errors.New("bytes: invalid offset")
48     }
49     if off >= int64(len(r.s)) {
50         return 0, io.EOF
51     }
52     n = copy(b, r.s[int(off):])
53     if n < len(b) {
54         err = io.EOF
55     }
56     return
57 }
58
59 func (r *Reader) ReadByte() (b byte, err error) {
60     if r.i >= len(r.s) {
61         return 0, io.EOF
62     }
63     b = r.s[r.i]
64     r.i++
65     r.prevRune = -1
66     return
67 }
68
69 func (r *Reader) UnreadByte() error {
70     if r.i <= 0 {
71         return errors.New("bytes.Reader: at beginning")
72     }
73     r.i--
74     r.prevRune = -1
75     return nil
76 }
77
78 func (r *Reader) ReadRune() (ch rune, size int, err error) {
79     if r.i >= len(r.s) {
80         return 0, 0, io.EOF
81     }
82     r.prevRune = r.i
83     if c := r.s[r.i]; c < utf8.RuneSelf {
84         r.i++
85         return rune(c), 1, nil
86     }
87     ch, size = utf8.DecodeRune(r.s[r.i:])
88     r.i += size
89     return
90 }
91
92 func (r *Reader) UnreadRune() error {
93     if r.prevRune < 0 {
94         return errors.New("bytes.Reader: previous op

```

```

95         }
96         r.i = r.prevRune
97         r.prevRune = -1
98         return nil
99     }
100
101     // Seek implements the io.Seeker interface.
102     func (r *Reader) Seek(offset int64, whence int) (int64, error) {
103         var abs int64
104         switch whence {
105             case 0:
106                 abs = offset
107             case 1:
108                 abs = int64(r.i) + offset
109             case 2:
110                 abs = int64(len(r.s)) + offset
111             default:
112                 return 0, errors.New("bytes: invalid whence")
113             }
114         if abs < 0 {
115             return 0, errors.New("bytes: negative position")
116         }
117         if abs >= 1<<31 {
118             return 0, errors.New("bytes: position out of range")
119         }
120         r.i = int(abs)
121         return abs, nil
122     }
123
124     // NewReader returns a new Reader reading from b.
125     func NewReader(b []byte) *Reader { return &Reader{b, 0, -1}

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/bzip2/bit_reader.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package bzip2
6
7 import (
8     "bufio"
9     "io"
10 )
11
12 // bitReader wraps an io.Reader and provides the ability to
13 // bit-by-bit, from it. Its Read* methods don't return the u
14 // because the error handling was verbose. Instead, any erro
15 // be checked afterwards.
16 type bitReader struct {
17     r    byteReader
18     n    uint64
19     bits uint
20     err  error
21 }
22
23 // bitReader needs to read bytes from an io.Reader. We attem
24 // given io.Reader to this interface and, if it doesn't alre
25 // a bufio.Reader.
26 type byteReader interface {
27     ReadByte() (byte, error)
28 }
29
30 func newBitReader(r io.Reader) bitReader {
31     byter, ok := r.(byteReader)
32     if !ok {
33         byter = bufio.NewReader(r)
34     }
35     return bitReader{r: byter}
36 }
37
38 // ReadBits64 reads the given number of bits and returns the
39 // least-significant part of a uint64. In the event of an er
40 // and the error can be obtained by calling Err().
41 func (br *bitReader) ReadBits64(bits uint) (n uint64) {
```

```

42     for bits > br.bits {
43         b, err := br.r.ReadByte()
44         if err == io.EOF {
45             err = io.ErrUnexpectedEOF
46         }
47         if err != nil {
48             br.err = err
49             return 0
50         }
51         br.n <<= 8
52         br.n |= uint64(b)
53         br.bits += 8
54     }
55
56     // br.n looks like this (assuming that br.bits = 14
57     // Bit: 111111
58     //      5432109876543210
59     //
60     //      (6 bits, the desired output)
61     //      |-----|
62     //      v       v
63     //      0101101101001110
64     //      ^             ^
65     //      |-----|
66     //      br.bits (num valid bits)
67     //
68     // This the next line right shifts the desired bits
69     // least-significant places and masks off anything a
70     n = (br.n >> (br.bits - bits)) & ((1 << bits) - 1)
71     br.bits -= bits
72     return
73 }
74
75 func (br *bitReader) ReadBits(bits uint) (n int) {
76     n64 := br.ReadBits64(bits)
77     return int(n64)
78 }
79
80 func (br *bitReader) ReadBit() bool {
81     n := br.ReadBits(1)
82     return n != 0
83 }
84
85 func (br *bitReader) Err() error {
86     return br.err
87 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/bzip2/bzip2.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package bzip2 implements bzip2 decompression.
6 package bzip2
7
8 import "io"
9
10 // There's no RFC for bzip2. I used the Wikipedia page for r
11 // of guessing: http://en.wikipedia.org/wiki/Bzip2
12 // The source code to pyflate was useful for debugging:
13 // http://www.paul.sladen.org/projects/pyflate
14
15 // A StructuralError is returned when the bzip2 data is found
16 // syntactically invalid.
17 type StructuralError string
18
19 func (s StructuralError) Error() string {
20     return "bzip2 data invalid: " + string(s)
21 }
22
23 // A reader decompresses bzip2 compressed data.
24 type reader struct {
25     br          bitReader
26     setupDone  bool // true if we have parsed the bzip2 header
27     blockSize  int  // blockSize in bytes, i.e. 900 * 1024
28     eof        bool
29     buf         []byte // stores Burrows-Wheeler transform
30     c           [256]uint // the 'C' array for the inverse transform
31     tt          [uint32] // mirrors the 'tt' array in the header
32     tPos        uint32 // Index of the next output byte
33
34     preRLE      [uint32] // contains the RLE data still to be read
35     preRLEUsed int // number of entries of preRLE that have been read
36     lastByte   int // the last byte value seen.
37     byteRepeats uint // the number of repeats of last byte
38     repeats    uint // the number of copies of last byte
39 }
40
41 // NewReader returns an io.Reader which decompresses bzip2 data.
```

```

42 func NewReader(r io.Reader) io.Reader {
43     bz2 := new(reader)
44     bz2.br = newBitReader(r)
45     return bz2
46 }
47
48 const bzip2FileMagic = 0x425a // "BZ"
49 const bzip2BlockMagic = 0x314159265359
50 const bzip2FinalMagic = 0x177245385090
51
52 // setup parses the bzip2 header.
53 func (bz2 *reader) setup() error {
54     br := &bz2.br
55
56     magic := br.ReadBits(16)
57     if magic != bzip2FileMagic {
58         return StructuralError("bad magic value")
59     }
60
61     t := br.ReadBits(8)
62     if t != 'h' {
63         return StructuralError("non-Huffman entropy")
64     }
65
66     level := br.ReadBits(8)
67     if level < '1' || level > '9' {
68         return StructuralError("invalid compression")
69     }
70
71     bz2.blockSize = 100 * 1024 * (int(level) - '0')
72     bz2.tt = make([]uint32, bz2.blockSize)
73     return nil
74 }
75
76 func (bz2 *reader) Read(buf []byte) (n int, err error) {
77     if bz2.eof {
78         return 0, io.EOF
79     }
80
81     if !bz2.setupDone {
82         err = bz2.setup()
83         brErr := bz2.br.Err()
84         if brErr != nil {
85             err = brErr
86         }
87         if err != nil {
88             return 0, err
89         }
90         bz2.setupDone = true
91     }

```

```

92
93     n, err = bz2.read(buf)
94     brErr := bz2.br.Err()
95     if brErr != nil {
96         err = brErr
97     }
98     return
99 }
100
101 func (bz2 *reader) read(buf []byte) (n int, err error) {
102     // bzip2 is a block based compressor, except that it
103     // preprocessing step. The block based nature means
104     // preallocate fixed-size buffers and reuse them. Ho
105     // preprocessing would require allocating huge buffe
106     // maximum expansion. Thus we process blocks all at
107     // the RLE which we decompress as required.
108
109     for (bz2.repeats > 0 || bz2.preRLEUsed < len(bz2.pre
110         // We have RLE data pending.
111
112         // The run-length encoding works like this:
113         // Any sequence of four equal bytes is follo
114         // byte which contains the number of repeats
115         // include. (The number of repeats can be ze
116         // decompressing on-demand our state is kept
117         // object.
118
119         if bz2.repeats > 0 {
120             buf[n] = byte(bz2.lastByte)
121             n++
122             bz2.repeats--
123             if bz2.repeats == 0 {
124                 bz2.lastByte = -1
125             }
126             continue
127         }
128
129         bz2.tPos = bz2.preRLE[bz2.tPos]
130         b := byte(bz2.tPos)
131         bz2.tPos >>= 8
132         bz2.preRLEUsed++
133
134         if bz2.byteRepeats == 3 {
135             bz2.repeats = uint(b)
136             bz2.byteRepeats = 0
137             continue
138         }
139
140         if bz2.lastByte == int(b) {

```

```

141             bz2.byteRepeats++
142         } else {
143             bz2.byteRepeats = 0
144         }
145         bz2.lastByte = int(b)
146
147         buf[n] = b
148         n++
149     }
150
151     if n > 0 {
152         return
153     }
154
155     // No RLE data is pending so we need to read a block
156
157     br := &bz2.br
158     magic := br.ReadBits64(48)
159     if magic == bzip2FinalMagic {
160         br.ReadBits64(32) // ignored CRC
161         bz2.eof = true
162         return 0, io.EOF
163     } else if magic != bzip2BlockMagic {
164         return 0, StructuralError("bad magic value f
165     }
166
167     err = bz2.readBlock()
168     if err != nil {
169         return 0, err
170     }
171
172     return bz2.read(buf)
173 }
174
175 // readBlock reads a bzip2 block. The magic number should al
176 func (bz2 *reader) readBlock() (err error) {
177     br := &bz2.br
178     br.ReadBits64(32) // skip checksum. TODO: check it i
179     randomized := br.ReadBits(1)
180     if randomized != 0 {
181         return StructuralError("deprecated randomize
182     }
183     origPtr := uint(br.ReadBits(24))
184
185     // If not every byte value is used in the block (i.e
186     // the symbol set is reduced. The symbols used are s
187     // two-level, 16x16 bitmap.
188     symbolRangeUsedBitmap := br.ReadBits(16)
189     symbolPresent := make([]bool, 256)

```

```

190     numSymbols := 0
191     for symRange := uint(0); symRange < 16; symRange++ {
192         if symbolRangeUsedBitmap&(1<<(15-symRange))
193             bits := br.ReadBits(16)
194             for symbol := uint(0); symbol < 16;
195                 if bits&(1<<(15-symbol)) !=
196                     symbolPresent[16*sym
197                         numSymbols++
198             }
199         }
200     }
201 }
202
203 // A block uses between two and six different Huffman
204 numHuffmanTrees := br.ReadBits(3)
205 if numHuffmanTrees < 2 || numHuffmanTrees > 6 {
206     return StructuralError("invalid number of Hu
207 }
208
209 // The Huffman tree can switch every 50 symbols so t
210 // tree indexes telling us which tree to use for eac
211 numSelectors := br.ReadBits(15)
212 treeIndexes := make([]uint8, numSelectors)
213
214 // The tree indexes are move-to-front transformed an
215 // numbers.
216 mtfTreeDecoder := newMTFDecoderWithRange(numHuffmanT
217 for i := range treeIndexes {
218     c := 0
219     for {
220         inc := br.ReadBits(1)
221         if inc == 0 {
222             break
223         }
224         c++
225     }
226     if c >= numHuffmanTrees {
227         return StructuralError("tree index t
228     }
229     treeIndexes[i] = uint8(mtfTreeDecoder.Decode
230 }
231
232 // The list of symbols for the move-to-front transfo
233 // the previously decoded symbol bitmap.
234 symbols := make([]byte, numSymbols)
235 nextSymbol := 0
236 for i := 0; i < 256; i++ {
237     if symbolPresent[i] {
238         symbols[nextSymbol] = byte(i)
239         nextSymbol++

```

```

240     }
241 }
242 mtf := newMTFDecoder(symbols)
243
244 numSymbols += 2 // to account for RUNA and RUNB symb
245 huffmanTrees := make([]huffmanTree, numHuffmanTrees)
246
247 // Now we decode the arrays of code-lengths for each
248 lengths := make([]uint8, numSymbols)
249 for i := 0; i < numHuffmanTrees; i++ {
250     // The code lengths are delta encoded from a
251     length := br.ReadBits(5)
252     for j := 0; j < numSymbols; j++ {
253         for {
254             if !br.ReadBit() {
255                 break
256             }
257             if br.ReadBit() {
258                 length--
259             } else {
260                 length++
261             }
262         }
263         if length < 0 || length > 20 {
264             return StructuralError("Huff
265         }
266         lengths[j] = uint8(length)
267     }
268     huffmanTrees[i], err = newHuffmanTree(length
269     if err != nil {
270         return err
271     }
272 }
273
274 selectorIndex := 1 // the next tree index to use
275 currentHuffmanTree := huffmanTrees[treeIndexes[0]]
276 bufIndex := 0 // indexes bz2.buf, the output buffer.
277 // The output of the move-to-front transform is run-
278 // we merge the decoding into the Huffman parsing lo
279 // variables accumulate the repeat count. See the wi
280 // details.
281 repeat := 0
282 repeat_power := 0
283
284 // The `C' array (used by the inverse BWT) needs to
285 for i := range bz2.c {
286     bz2.c[i] = 0
287 }
288

```

```

289     decoded := 0 // counts the number of symbols decoded
290     for {
291         if decoded == 50 {
292             currentHuffmanTree = huffmanTrees[tr
293                 selectorIndex++
294                 decoded = 0
295         }
296
297         v := currentHuffmanTree.Decode(br)
298         decoded++
299
300         if v < 2 {
301             // This is either the RUNA or RUNB s
302             if repeat == 0 {
303                 repeat_power = 1
304             }
305             repeat += repeat_power << v
306             repeat_power <<= 1
307
308             // This limit of 2 million comes fro
309             // code. It prevents repeat from ove
310             if repeat > 2*1024*1024 {
311                 return StructuralError("repe
312             }
313             continue
314         }
315
316         if repeat > 0 {
317             // We have decoded a complete run-le
318             // replicate the last output symbol.
319             for i := 0; i < repeat; i++ {
320                 b := byte(mtf.First())
321                 bz2.tt[bufIndex] = uint32(b)
322                 bz2.c[b]++
323                 bufIndex++
324             }
325             repeat = 0
326         }
327
328         if int(v) == numSymbols-1 {
329             // This is the EOF symbol. Because i
330             // end of the move-to-front list, an
331             // to the front, it has this unique
332             break
333         }
334
335         // Since two metasymbols (RUNA and RUNB) hav
336         // one would expect |v-2| to be passed to th
337         // However, the front of the MTF list is nev

```

```

338         // it's always referenced with a run-length
339         // doesn't need to be encoded and we have |v
340         // line.
341         b := byte(mtf.Decode(int(v - 1)))
342         bz2.tt[bufIndex] = uint32(b)
343         bz2.c[b]++
344         bufIndex++
345     }
346
347     if origPtr >= uint(bufIndex) {
348         return StructuralError("origPtr out of bound
349     }
350
351     // We have completed the entropy decoding. Now we ca
352     // inverse BWT and setup the RLE buffer.
353     bz2.preRLE = bz2.tt[:bufIndex]
354     bz2.preRLEUsed = 0
355     bz2.tPos = inverseBWT(bz2.preRLE, origPtr, bz2.c[:])
356     bz2.lastByte = -1
357     bz2.byteRepeats = 0
358     bz2.repeats = 0
359
360     return nil
361 }
362
363 // inverseBWT implements the inverse Burrows-Wheeler transfo
364 // http://www.hp1.hp.com/techreports/Compaq-DEC/SRC-RR-124.p
365 // In that document, origPtr is called `I' and c is the `C'
366 // first pass over the data. It's an argument here because w
367 // pass with the Huffman decoding.
368 //
369 // This also implements the `single array' method from the b
370 // which leaves the output, still shuffled, in the bottom 8
371 // index of the next byte in the top 24-bits. The index of t
372 // returned.
373 func inverseBWT(tt []uint32, origPtr uint, c []uint) uint32
374     sum := uint(0)
375     for i := 0; i < 256; i++ {
376         sum += c[i]
377         c[i] = sum - c[i]
378     }
379
380     for i := range tt {
381         b := tt[i] & 0xff
382         tt[c[b]] |= uint32(i) << 8
383         c[b]++
384     }
385
386     return tt[origPtr] >> 8
387 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/bzip2/huffman.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package bzip2
6
7 import "sort"
8
9 // A huffmanTree is a binary tree which is navigated, bit-by-
10 // symbol.
11 type huffmanTree struct {
12     // nodes contains all the non-leaf nodes in the tree
13     // root of the tree and nextNode contains the index
14     // of nodes to use when the tree is being constructed
15     nodes    []huffmanNode
16     nextNode int
17 }
18
19 // A huffmanNode is a node in the tree. left and right contain
20 // nodes slice of the tree. If left or right is invalidNodeValue
21 // is a left node and its value is in leftValue/rightValue.
22 //
23 // The symbols are uint16s because bzip2 encodes not only MT
24 // tree, but also two magic values for run-length encoding a
25 // Thus there are more than 256 possible symbols.
26 type huffmanNode struct {
27     left, right          uint16
28     leftValue, rightValue uint16
29 }
30
31 // invalidNodeValue is an invalid index which marks a leaf node
32 const invalidNodeValue = 0xffff
33
34 // Decode reads bits from the given bitReader and navigates
35 // symbol is found.
36 func (t huffmanTree) Decode(br *bitReader) (v uint16) {
37     nodeIndex := uint16(0) // node 0 is the root of the
38
39     for {
40         node := &t.nodes[nodeIndex]
41         bit := br.ReadBit()
```

```

42         // bzip2 encodes left as a true bit.
43         if bit {
44             // left
45             if node.left == invalidNodeValue {
46                 return node.leftValue
47             }
48             nodeIndex = node.left
49         } else {
50             // right
51             if node.right == invalidNodeValue {
52                 return node.rightValue
53             }
54             nodeIndex = node.right
55         }
56     }
57     panic("unreachable")
58 }
59 }
60
61 // newHuffmanTree builds a Huffman tree from a slice contain
62 // lengths of each symbol. The maximum code length is 32 bit
63 func newHuffmanTree(lengths []uint8) (huffmanTree, error) {
64     // There are many possible trees that assign the same
65     // each symbol (consider reflecting a tree down the
66     // example). Since the code length assignments deter
67     // efficiency of the tree, each of these trees is eq
68     // order to minimize the amount of information neede
69     // bzip2 uses a canonical tree so that it can be rec
70     // only the code length assignments.
71
72     if len(lengths) < 2 {
73         panic("newHuffmanTree: too few symbols")
74     }
75
76     var t huffmanTree
77
78     // First we sort the code length assignments by asce
79     // using the symbol value to break ties.
80     pairs := huffmanSymbolLengthPairs(make([]huffmanSymb
81     for i, length := range lengths {
82         pairs[i].value = uint16(i)
83         pairs[i].length = length
84     }
85
86     sort.Sort(pairs)
87
88     // Now we assign codes to the symbols, starting with
89     // We keep the codes packed into a uint32, at the mo
90     // So branches are taken from the MSB downwards. Thi
91     // sort them later.

```

```

92     code := uint32(0)
93     length := uint8(32)
94
95     codes := huffmanCodes(make([]huffmanCode, len(length)
96     for i := len(pairs) - 1; i >= 0; i-- {
97         if length > pairs[i].length {
98             // If the code length decreases we s
99             // zero any bits beyond the end of t
100            length >>= 32 - pairs[i].length
101            length <<= 32 - pairs[i].length
102            length = pairs[i].length
103        }
104        codes[i].code = code
105        codes[i].codeLen = length
106        codes[i].value = pairs[i].value
107        // We need to 'increment' the code, which me
108        // like a |length| bit number.
109        code += 1 << (32 - length)
110    }
111
112    // Now we can sort by the code so that the left half
113    // grouped together, recursively.
114    sort.Sort(codes)
115
116    t.nodes = make([]huffmanNode, len(codes))
117    _, err := buildHuffmanNode(&t, codes, 0)
118    return t, err
119 }
120
121 // huffmanSymbolLengthPair contains a symbol and its code le
122 type huffmanSymbolLengthPair struct {
123     value uint16
124     length uint8
125 }
126
127 // huffmanSymbolLengthPair is used to provide an interface f
128 type huffmanSymbolLengthPairs []huffmanSymbolLengthPair
129
130 func (h huffmanSymbolLengthPairs) Len() int {
131     return len(h)
132 }
133
134 func (h huffmanSymbolLengthPairs) Less(i, j int) bool {
135     if h[i].length < h[j].length {
136         return true
137     }
138     if h[i].length > h[j].length {
139         return false
140     }

```

```

141         if h[i].value < h[j].value {
142             return true
143         }
144         return false
145     }
146
147     func (h huffmanSymbolLengthPairs) Swap(i, j int) {
148         h[i], h[j] = h[j], h[i]
149     }
150
151     // huffmanCode contains a symbol, its code and code length.
152     type huffmanCode struct {
153         code    uint32
154         codeLen uint8
155         value   uint16
156     }
157
158     // huffmanCodes is used to provide an interface for sorting.
159     type huffmanCodes []huffmanCode
160
161     func (n huffmanCodes) Len() int {
162         return len(n)
163     }
164
165     func (n huffmanCodes) Less(i, j int) bool {
166         return n[i].code < n[j].code
167     }
168
169     func (n huffmanCodes) Swap(i, j int) {
170         n[i], n[j] = n[j], n[i]
171     }
172
173     // buildHuffmanNode takes a slice of sorted huffmanCodes and
174     // the Huffman tree at the given level. It returns the index
175     // constructed node.
176     func buildHuffmanNode(t *huffmanTree, codes []huffmanCode, l
177         test := uint32(1) << (31 - level)
178
179         // We have to search the list of codes to find the d
180         firstRightIndex := len(codes)
181         for i, code := range codes {
182             if code.code&test != 0 {
183                 firstRightIndex = i
184                 break
185             }
186         }
187
188         left := codes[:firstRightIndex]
189         right := codes[firstRightIndex:]

```

```

190
191     if len(left) == 0 || len(right) == 0 {
192         return 0, StructuralError("superfluous level
193     }
194
195     nodeIndex = uint16(t.nextNode)
196     node := &t.nodes[t.nextNode]
197     t.nextNode++
198
199     if len(left) == 1 {
200         // leaf node
201         node.left = invalidNodeValue
202         node.leftValue = left[0].value
203     } else {
204         node.left, err = buildHuffmanNode(t, left, 1
205     }
206
207     if err != nil {
208         return
209     }
210
211     if len(right) == 1 {
212         // leaf node
213         node.right = invalidNodeValue
214         node.rightValue = right[0].value
215     } else {
216         node.right, err = buildHuffmanNode(t, right,
217     }
218
219     return
220 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/bzip2/move_to_front

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package bzip2
6
7 // moveToFrontDecoder implements a move-to-front list. Such
8 // efficient way to transform a string with repeating elemen
9 // many small valued numbers, which is suitable for entropy
10 // by starting with an initial list of symbols and reference
11 // index into that list. When a symbol is referenced, it's m
12 // of the list. Thus, a repeated symbol ends up being encode
13 // as the symbol will be at the front of the list after the
14 type moveToFrontDecoder struct {
15     // Rather than actually keep the list in memory, the
16     // as a circular, double linked list with the symbol
17     // at the front of the list.
18     symbols []byte
19     next    []uint8
20     prev    []uint8
21     head    uint8
22 }
23
24 // newMTFDecoder creates a move-to-front decoder with an exp
25 // of symbols.
26 func newMTFDecoder(symbols []byte) *moveToFrontDecoder {
27     if len(symbols) > 256 {
28         panic("too many symbols")
29     }
30
31     m := &moveToFrontDecoder{
32         symbols: symbols,
33         next:    make([]uint8, len(symbols)),
34         prev:    make([]uint8, len(symbols)),
35     }
36
37     m.threadLinkedList()
38     return m
39 }
40
41 // newMTFDecoderWithRange creates a move-to-front decoder wi
```

```

42 // symbol list of 0...n-1.
43 func newMTFDecoderWithRange(n int) *moveToFrontDecoder {
44     if n > 256 {
45         panic("newMTFDecoderWithRange: cannot have >
46     }
47
48     m := &moveToFrontDecoder{
49         symbols: make([]uint8, n),
50         next:    make([]uint8, n),
51         prev:    make([]uint8, n),
52     }
53
54     for i := 0; i < n; i++ {
55         m.symbols[i] = byte(i)
56     }
57
58     m.threadLinkedList()
59     return m
60 }
61
62 // threadLinkedList creates the initial linked-list pointers
63 func (m *moveToFrontDecoder) threadLinkedList() {
64     if len(m.symbols) == 0 {
65         return
66     }
67
68     m.prev[0] = uint8(len(m.symbols) - 1)
69
70     for i := 0; i < len(m.symbols)-1; i++ {
71         m.next[i] = uint8(i + 1)
72         m.prev[i+1] = uint8(i)
73     }
74
75     m.next[len(m.symbols)-1] = 0
76 }
77
78 func (m *moveToFrontDecoder) Decode(n int) (b byte) {
79     // Most of the time, n will be zero so it's worth de
80     // simple case.
81     if n == 0 {
82         return m.symbols[m.head]
83     }
84
85     i := m.head
86     for j := 0; j < n; j++ {
87         i = m.next[i]
88     }
89     b = m.symbols[i]
90
91     m.next[m.prev[i]] = m.next[i]

```

```
92         m.prev[m.next[i]] = m.prev[i]
93         m.next[i] = m.head
94         m.prev[i] = m.prev[m.head]
95         m.next[m.prev[m.head]] = i
96         m.prev[m.head] = i
97         m.head = i
98
99         return
100    }
101
102    // First returns the symbol at the front of the list.
103    func (m *moveToFrontDecoder) First() byte {
104        return m.symbols[m.head]
105    }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/flate/deflate.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package flate
6
7 import (
8     "fmt"
9     "io"
10    "math"
11 )
12
13 const (
14     NoCompression      = 0
15     BestSpeed          = 1
16     fastCompression    = 3
17     BestCompression    = 9
18     DefaultCompression = -1
19     logWindowSize      = 15
20     windowSize         = 1 << logWindowSize
21     windowMask        = windowSize - 1
22     logMaxOffsetSize   = 15 // Standard DEFLATE
23     minMatchLength     = 3  // The smallest match that
24     maxMatchLength     = 258 // The longest match for th
25     minOffsetSize      = 1  // The shortest offset that
26
27     // The maximum number of tokens we put into a single
28     // stop things from getting too large.
29     maxFlateBlockTokens = 1 << 14
30     maxStoreBlockSize   = 65535
31     hashBits            = 17
32     hashSize            = 1 << hashBits
33     hashMask           = (1 << hashBits) - 1
34     hashShift          = (hashBits + minMatchLength - 1
35
36     skipNever = math.MaxInt32
37 )
38
39 type compressionLevel struct {
40     good, lazy, nice, chain, fastSkipHashing int
41 }
```

```

42
43 var levels = []compressionLevel{
44     {}, // 0
45     // For levels 1-3 we don't bother trying with lazy m
46     {3, 0, 8, 4, 4},
47     {3, 0, 16, 8, 5},
48     {3, 0, 32, 32, 6},
49     // Levels 4-9 use increasingly more lazy matching
50     // and increasingly stringent conditions for "good e
51     {4, 4, 16, 16, skipNever},
52     {8, 16, 32, 32, skipNever},
53     {8, 16, 128, 128, skipNever},
54     {8, 32, 128, 256, skipNever},
55     {32, 128, 258, 1024, skipNever},
56     {32, 258, 258, 4096, skipNever},
57 }
58
59 type compressor struct {
60     compressionLevel
61
62     w *huffmanBitWriter
63
64     // compression algorithm
65     fill func(*compressor, []byte) int // copy data to w
66     step func(*compressor)           // process window
67     sync bool                         // requesting flu
68
69     // Input hash chains
70     // hashHead[hashValue] contains the largest inputInd
71     // If hashHead[hashValue] is within the current wind
72     // hashPrev[hashHead[hashValue] & windowMask] contai
73     // with the same hash value.
74     chainHead int
75     hashHead []int
76     hashPrev []int
77     hashOffset int
78
79     // input window: unprocessed data is window[index:wi
80     index int
81     window []byte
82     windowEnd int
83     blockStart int // window index where current tok
84     byteAvailable bool // if true, still need to process
85
86     // queued output tokens
87     tokens []token
88
89     // deflate state
90     length int
91     offset int

```

```

92         hash          int
93         maxInsertIndex int
94         err           error
95     }
96
97     func (d *compressor) fillDeflate(b []byte) int {
98         if d.index >= 2*windowSize-(minMatchLength+maxMatchL
99             // shift the window by windowSize
100            copy(d.window, d.window[windowSize:2*windowS
101            d.index -= windowSize
102            d.windowEnd -= windowSize
103            if d.blockStart >= windowSize {
104                d.blockStart -= windowSize
105            } else {
106                d.blockStart = math.MaxInt32
107            }
108            d.hashOffset += windowSize
109        }
110        n := copy(d.window[d.windowEnd:], b)
111        d.windowEnd += n
112        return n
113    }
114
115     func (d *compressor) writeBlock(tokens []token, index int, e
116         if index > 0 || eof {
117             var window []byte
118             if d.blockStart <= index {
119                 window = d.window[d.blockStart:index
120             }
121             d.blockStart = index
122             d.w.writeBlock(tokens, eof, window)
123             return d.w.err
124         }
125         return nil
126     }
127
128     // Try to find a match starting at index whose length is gre
129     // We only look at chainCount possibilities before giving up
130     func (d *compressor) findMatch(pos int, prevHead int, prevLe
131         minMatchLook := maxMatchLength
132         if lookahead < minMatchLook {
133             minMatchLook = lookahead
134         }
135
136         win := d.window[0 : pos+minMatchLook]
137
138         // We quit when we get a match that's at least nice
139         nice := len(win) - pos
140         if d.nice < nice {

```

```

141         nice = d.nice
142     }
143
144     // If we've got a match that's good enough, only loo
145     tries := d.chain
146     length = prevLength
147     if length >= d.good {
148         tries >>= 2
149     }
150
151     w0 := win[pos]
152     w1 := win[pos+1]
153     wEnd := win[pos+length]
154     minIndex := pos - windowSize
155
156     for i := prevHead; tries > 0; tries-- {
157         if w0 == win[i] && w1 == win[i+1] && wEnd ==
158             // The hash function ensures that if
159
160             n := 3
161             for pos+n < len(win) && win[i+n] ==
162                 n++
163             }
164             if n > length && (n > 3 || pos-i <=
165                 length = n
166                 offset = pos - i
167                 ok = true
168                 if n >= nice {
169                     // The match is good
170                     break
171                 }
172                 wEnd = win[pos+n]
173             }
174         }
175         if i == minIndex {
176             // hashPrev[i & windowMask] has alre
177             break
178         }
179         if i = d.hashPrev[i&windowMask] - d.hashOffs
180             break
181     }
182     }
183     return
184 }
185
186 func (d *compressor) writeStoredBlock(buf []byte) error {
187     if d.w.writeStoredHeader(len(buf), false); d.w.err !
188         return d.w.err
189     }

```

```

190         d.w.writeBytes(buf)
191         return d.w.err
192     }
193
194     func (d *compressor) initDeflate() {
195         d.hashHead = make([]int, hashSize)
196         d.hashPrev = make([]int, windowSize)
197         d.window = make([]byte, 2*windowSize)
198         d.hashOffset = 1
199         d.tokens = make([]token, 0, maxFlateBlockTokens+1)
200         d.length = minMatchLength - 1
201         d.offset = 0
202         d.byteAvailable = false
203         d.index = 0
204         d.hash = 0
205         d.chainHead = -1
206     }
207
208     func (d *compressor) deflate() {
209         if d.windowEnd-d.index < minMatchLength+maxMatchLeng
210             return
211     }
212
213     d.maxInsertIndex = d.windowEnd - (minMatchLength - 1
214     if d.index < d.maxInsertIndex {
215         d.hash = int(d.window[d.index])<<hashShift +
216     }
217
218     Loop:
219     for {
220         if d.index > d.windowEnd {
221             panic("index > windowEnd")
222         }
223         lookahead := d.windowEnd - d.index
224         if lookahead < minMatchLength+maxMatchLength
225             if !d.sync {
226                 break Loop
227             }
228             if d.index > d.windowEnd {
229                 panic("index > windowEnd")
230             }
231             if lookahead == 0 {
232                 // Flush current output bloc
233                 if d.byteAvailable {
234                     // There is still on
235                     d.tokens = append(d.
236                     d.byteAvailable = fa
237                 }
238                 if len(d.tokens) > 0 {
239                     if d.err = d.writeBl

```

```

240                                     return
241                                     }
242                                     d.tokens = d.tokens[
243                                     ]
244                                     break Loop
245                                 }
246                             }
247                             if d.index < d.maxInsertIndex {
248                                 // Update the hash
249                                 d.hash = (d.hash<<hashShift + int(d.
250                                 d.chainHead = d.hashHead[d.hash]
251                                 d.hashPrev[d.index&windowMask] = d.c
252                                 d.hashHead[d.hash] = d.index + d.has
253                             }
254                             prevLength := d.length
255                             prevOffset := d.offset
256                             d.length = minMatchLength - 1
257                             d.offset = 0
258                             minIndex := d.index - windowSize
259                             if minIndex < 0 {
260                                 minIndex = 0
261                             }
262
263                             if d.chainHead-d.hashOffset >= minIndex &&
264                                 (d.fastSkipHashing != skipNever && 1
265                                 d.fastSkipHashing == skipNev
266                                 if newLength, newOffset, ok := d.fin
267                                 d.length = newLength
268                                 d.offset = newOffset
269                             }
270                         }
271                         if d.fastSkipHashing != skipNever && d.lengt
272                         d.fastSkipHashing == skipNever && pr
273                         // There was a match at the previous
274                         // not better. Output the previous m
275                         if d.fastSkipHashing != skipNever {
276                             d.tokens = append(d.tokens,
277                         } else {
278                             d.tokens = append(d.tokens,
279                         }
280                         // Insert in the hash table all stri
281                         // index and index-1 are already ins
282                         // lookahead, the last two strings a
283                         // table.
284                         if d.length <= d.fastSkipHashing {
285                             var newIndex int
286                             if d.fastSkipHashing != skip
287                                 newIndex = d.index +
288                         } else {

```

```

289             newIndex = d.index +
290         }
291         for d.index++; d.index < new
292             if d.index < d.maxIn
293                 d.hash = (d.
294                     // Get previ
295                     // Our chain
296                     d.hashPrev[d
297                     // Set the h
298                     d.hashHead[d
299                 }
300         }
301         if d.fastSkipHashing == skip
302             d.byteAvailable = fa
303             d.length = minMatchL
304     }
305 } else {
306     // For matches this long, we
307     // item into the table.
308     d.index += d.length
309     if d.index < d.maxInsertInde
310         d.hash = (int(d.wind
311     }
312 }
313 if len(d.tokens) == maxFlateBlockTok
314     // The block includes the cu
315     if d.err = d.writeBlock(d.to
316         return
317     }
318     d.tokens = d.tokens[:0]
319 }
320 } else {
321     if d.fastSkipHashing != skipNever ||
322         i := d.index - 1
323         if d.fastSkipHashing != skip
324             i = d.index
325     }
326     d.tokens = append(d.tokens,
327         if len(d.tokens) == maxFlate
328             if d.err = d.writeBl
329                 return
330     }
331     d.tokens = d.tokens[
332     }
333 }
334 d.index++
335 if d.fastSkipHashing == skipNever {
336     d.byteAvailable = true
337 }

```

```

338         }
339     }
340 }
341
342 func (d *compressor) fillStore(b []byte) int {
343     n := copy(d.window[d.windowEnd:], b)
344     d.windowEnd += n
345     return n
346 }
347
348 func (d *compressor) store() {
349     if d.windowEnd > 0 {
350         d.err = d.writeStoredBlock(d.window[:d.windowEnd])
351     }
352     d.windowEnd = 0
353 }
354
355 func (d *compressor) write(b []byte) (n int, err error) {
356     n = len(b)
357     b = b[d.fill(d, b):]
358     for len(b) > 0 {
359         d.step(d)
360         b = b[d.fill(d, b):]
361     }
362     return n, d.err
363 }
364
365 func (d *compressor) syncFlush() error {
366     d.sync = true
367     d.step(d)
368     if d.err == nil {
369         d.w.writeStoredHeader(0, false)
370         d.w.flush()
371         d.err = d.w.err
372     }
373     d.sync = false
374     return d.err
375 }
376
377 func (d *compressor) init(w io.Writer, level int) (err error) {
378     d.w = newHuffmanBitWriter(w)
379
380     switch {
381     case level == NoCompression:
382         d.window = make([]byte, maxStoreBlockSize)
383         d.fill = (*compressor).fillStore
384         d.step = (*compressor).store
385     case level == DefaultCompression:
386         level = 6
387         fallthrough

```

```

388         case 1 <= level && level <= 9:
389             d.compressionLevel = levels[level]
390             d.initDeflate()
391             d.fill = (*compressor).fillDeflate
392             d.step = (*compressor).deflate
393         default:
394             return fmt.Errorf("flate: invalid compressio
395     }
396     return nil
397 }
398
399 func (d *compressor) close() error {
400     d.sync = true
401     d.step(d)
402     if d.err != nil {
403         return d.err
404     }
405     if d.w.writeStoredHeader(0, true); d.w.err != nil {
406         return d.w.err
407     }
408     d.w.flush()
409     return d.w.err
410 }
411
412 // NewWriter returns a new Writer compressing data at the gi
413 // Following zlib, levels range from 1 (BestSpeed) to 9 (Bes
414 // higher levels typically run slower but compress more. Lev
415 // (NoCompression) does not attempt any compression; it only
416 // necessary DEFLATE framing. Level -1 (DefaultCompression)
417 // compression level.
418 //
419 // If level is in the range [-1, 9] then the error returned
420 // Otherwise the error returned will be non-nil.
421 func NewWriter(w io.Writer, level int) (*Writer, error) {
422     const logWindowSize = logMaxOffsetSize
423     var dw Writer
424     if err := dw.d.init(w, level); err != nil {
425         return nil, err
426     }
427     return &dw, nil
428 }
429
430 // NewWriterDict is like NewWriter but initializes the new
431 // Writer with a preset dictionary. The returned Writer beh
432 // as if the dictionary had been written to it without produ
433 // any compressed output. The compressed data written to w
434 // can only be decompressed by a Reader initialized with the
435 // same dictionary.
436 func NewWriterDict(w io.Writer, level int, dict []byte) (*Wr

```

```

437         dw := &dictWriter{w, false}
438         zw, err := NewWriter(dw, level)
439         if err != nil {
440             return nil, err
441         }
442         zw.Write(dict)
443         zw.Flush()
444         dw.enabled = true
445         return zw, err
446     }
447
448     type dictWriter struct {
449         w      io.Writer
450         enabled bool
451     }
452
453     func (w *dictWriter) Write(b []byte) (n int, err error) {
454         if w.enabled {
455             return w.w.Write(b)
456         }
457         return len(b), nil
458     }
459
460     // A Writer takes data written to it and writes the compress
461     // form of that data to an underlying writer (see NewWriter)
462     type Writer struct {
463         d compressor
464     }
465
466     // Write writes data to w, which will eventually write the
467     // compressed form of data to its underlying writer.
468     func (w *Writer) Write(data []byte) (n int, err error) {
469         return w.d.write(data)
470     }
471
472     // Flush flushes any pending compressed data to the underlyi
473     // It is useful mainly in compressed network protocols, to e
474     // a remote reader has enough data to reconstruct a packet.
475     // Flush does not return until the data has been written.
476     // If the underlying writer returns an error, Flush returns
477     //
478     // In the terminology of the zlib library, Flush is equivale
479     func (w *Writer) Flush() error {
480         // For more about flushing:
481         // http://www.bolet.org/~pornin/deflate-flush.html
482         return w.d.syncFlush()
483     }
484
485     // Close flushes and closes the writer.

```

```
486 func (w *Writer) Close() error {  
487     return w.d.close()  
488 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/flate/huffman_bit_w

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package flate
6
7 import (
8     "io"
9     "math"
10 )
11
12 const (
13     // The largest offset code.
14     offsetCodeCount = 30
15
16     // The special code used to mark the end of a block.
17     endBlockMarker = 256
18
19     // The first length code.
20     lengthCodesStart = 257
21
22     // The number of codegen codes.
23     codegenCodeCount = 19
24     badCode           = 255
25 )
26
27 // The number of extra bits needed by length code X - LENGTH
28 var lengthExtraBits = []int8{
29     /* 257 */ 0, 0, 0,
30     /* 260 */ 0, 0, 0, 0, 0, 1, 1, 1, 1, 2,
31     /* 270 */ 2, 2, 2, 3, 3, 3, 3, 4, 4, 4,
32     /* 280 */ 4, 5, 5, 5, 5, 0,
33 }
34
35 // The length indicated by length code X - LENGTH_CODES_STAR
36 var lengthBase = []uint32{
37     0, 1, 2, 3, 4, 5, 6, 7, 8, 10,
38     12, 14, 16, 20, 24, 28, 32, 40, 48, 56,
39     64, 80, 96, 112, 128, 160, 192, 224, 255,
40 }
41
```

```

42 // offset code word extra bits.
43 var offsetExtraBits = []int8{
44     0, 0, 0, 0, 1, 1, 2, 2, 3, 3,
45     4, 4, 5, 5, 6, 6, 7, 7, 8, 8,
46     9, 9, 10, 10, 11, 11, 12, 12, 13, 13,
47     /* extended window */
48     14, 14, 15, 15, 16, 16, 17, 17, 18, 18, 19, 19, 20,
49 }
50
51 var offsetBase = []uint32{
52     /* normal deflate */
53     0x000000, 0x000001, 0x000002, 0x000003, 0x000004,
54     0x000006, 0x000008, 0x00000c, 0x000010, 0x000018,
55     0x000020, 0x000030, 0x000040, 0x000060, 0x000080,
56     0x0000c0, 0x000100, 0x000180, 0x000200, 0x000300,
57     0x000400, 0x000600, 0x000800, 0x000c00, 0x001000,
58     0x001800, 0x002000, 0x003000, 0x004000, 0x006000,
59
60     /* extended window */
61     0x008000, 0x00c000, 0x010000, 0x018000, 0x020000,
62     0x030000, 0x040000, 0x060000, 0x080000, 0x0c0000,
63     0x100000, 0x180000, 0x200000, 0x300000,
64 }
65
66 // The odd order in which the codegen code sizes are written
67 var codegenOrder = []uint32{16, 17, 18, 0, 8, 7, 9, 6, 10, 5
68
69 type huffmanBitWriter struct {
70     w io.Writer
71     // Data waiting to be written is bytes[0:nbytes]
72     // and then the low nbits of bits.
73     bits          uint32
74     nbits         uint32
75     bytes        [64]byte
76     nbytes       int
77     literalFreq  []int32
78     offsetFreq   []int32
79     codegen      []uint8
80     codegenFreq  []int32
81     literalEncoding *huffmanEncoder
82     offsetEncoding *huffmanEncoder
83     codegenEncoding *huffmanEncoder
84     err           error
85 }
86
87 func newHuffmanBitWriter(w io.Writer) *huffmanBitWriter {
88     return &huffmanBitWriter{
89         w:          w,
90         literalFreq: make([]int32, maxLit),
91         offsetFreq:  make([]int32, offsetCodeCou

```

```

92             codegen:          make([]uint8, maxLit+offset
93             codegenFreq:      make([]int32, codegenCodeCo
94             literalEncoding:  newHuffmanEncoder(maxLit),
95             offsetEncoding:   newHuffmanEncoder(offsetCod
96             codegenEncoding:  newHuffmanEncoder(codegenCo
97         }
98     }
99
100 func (w *huffmanBitWriter) flushBits() {
101     if w.err != nil {
102         w.nbits = 0
103         return
104     }
105     bits := w.bits
106     w.bits >>= 16
107     w.nbits -= 16
108     n := w.nbytes
109     w.bytes[n] = byte(bits)
110     w.bytes[n+1] = byte(bits >> 8)
111     if n += 2; n >= len(w.bytes) {
112         _, w.err = w.w.Write(w.bytes[0:])
113         n = 0
114     }
115     w.nbytes = n
116 }
117
118 func (w *huffmanBitWriter) flush() {
119     if w.err != nil {
120         w.nbits = 0
121         return
122     }
123     n := w.nbytes
124     if w.nbits > 8 {
125         w.bytes[n] = byte(w.bits)
126         w.bits >>= 8
127         w.nbits -= 8
128         n++
129     }
130     if w.nbits > 0 {
131         w.bytes[n] = byte(w.bits)
132         w.nbits = 0
133         n++
134     }
135     w.bits = 0
136     _, w.err = w.w.Write(w.bytes[0:n])
137     w.nbytes = 0
138 }
139
140 func (w *huffmanBitWriter) writeBits(b, nb int32) {

```

```

141         w.bits |= uint32(b) << w.nbits
142         if w.nbits += uint32(nb); w.nbits >= 16 {
143             w.flushBits()
144         }
145     }
146
147     func (w *huffmanBitWriter) writeBytes(bytes []byte) {
148         if w.err != nil {
149             return
150         }
151         n := w.nbytes
152         if w.nbits == 8 {
153             w.bytes[n] = byte(w.bits)
154             w.nbits = 0
155             n++
156         }
157         if w.nbits != 0 {
158             w.err = InternalError("writeBytes with unfin
159             return
160         }
161         if n != 0 {
162             _, w.err = w.w.Write(w.bytes[0:n])
163             if w.err != nil {
164                 return
165             }
166         }
167         w.nbytes = n
168         _, w.err = w.w.Write(bytes)
169     }
170
171     // RFC 1951 3.2.7 specifies a special run-length encoding fo
172     // the literal and offset lengths arrays (which are concaten
173     // array). This method generates that run-length encoding.
174     //
175     // The result is written into the codegen array, and the fre
176     // of each code is written into the codegenFreq array.
177     // Codes 0-15 are single byte codes. Codes 16-18 are followe
178     // information. Code badCode is an end marker
179     //
180     // numLiterals      The number of literals in literalEncodi
181     // numOffsets       The number of offsets in offsetEncoding
182     func (w *huffmanBitWriter) generateCodegen(numLiterals int,
183         for i := range w.codegenFreq {
184             w.codegenFreq[i] = 0
185         }
186         // Note that we are using codegen both as a temporar
187         // a copy of the frequencies, and as the place where
188         // This is fine because the output is always shorter
189         // so far.

```

```

190     codegen := w.codegen // cache
191     // Copy the concatenated code sizes to codegen. Put
192     copy(codegen[0:numLiterals], w.literalEncoding.codeB
193     copy(codegen[numLiterals:numLiterals+numOffsets], w.
194     codegen[numLiterals+numOffsets] = badCode
195
196     size := codegen[0]
197     count := 1
198     outIndex := 0
199     for inIndex := 1; size != badCode; inIndex++ {
200         // INVARIANT: We have seen "count" copies of
201         // had output generated for them.
202         nextSize := codegen[inIndex]
203         if nextSize == size {
204             count++
205             continue
206         }
207         // We need to generate codegen indicating "c
208         if size != 0 {
209             codegen[outIndex] = size
210             outIndex++
211             w.codegenFreq[size]++
212             count--
213             for count >= 3 {
214                 n := 6
215                 if n > count {
216                     n = count
217                 }
218                 codegen[outIndex] = 16
219                 outIndex++
220                 codegen[outIndex] = uint8(n
221                 outIndex++
222                 w.codegenFreq[16]++
223                 count -= n
224             }
225         } else {
226             for count >= 11 {
227                 n := 138
228                 if n > count {
229                     n = count
230                 }
231                 codegen[outIndex] = 18
232                 outIndex++
233                 codegen[outIndex] = uint8(n
234                 outIndex++
235                 w.codegenFreq[18]++
236                 count -= n
237             }
238             if count >= 3 {
239                 // count >= 3 && count <= 10

```

```

240         codegen[outIndex] = 17
241         outIndex++
242         codegen[outIndex] = uint8(cc
243         outIndex++
244         w.codegenFreq[17]++
245         count = 0
246     }
247 }
248 count--
249 for ; count >= 0; count-- {
250     codegen[outIndex] = size
251     outIndex++
252     w.codegenFreq[size]++
253 }
254 // Set up invariant for next time through th
255 size = nextSize
256 count = 1
257 }
258 // Marker indicating the end of the codegen.
259 codegen[outIndex] = badCode
260 }
261
262 func (w *huffmanBitWriter) writeCode(code *huffmanEncoder, l
263     if w.err != nil {
264         return
265     }
266     w.writeBits(int32(code.code[literal]), int32(code.co
267 }
268
269 // Write the header of a dynamic Huffman block to the output
270 //
271 // numLiterals The number of literals specified in codegen
272 // numOffsets The number of offsets specified in codegen
273 // numCodegens The number of codegens used in codegen
274 func (w *huffmanBitWriter) writeDynamicHeader(numLiterals in
275     if w.err != nil {
276         return
277     }
278     var firstBits int32 = 4
279     if isEof {
280         firstBits = 5
281     }
282     w.writeBits(firstBits, 3)
283     w.writeBits(int32(numLiterals-257), 5)
284     w.writeBits(int32(numOffsets-1), 5)
285     w.writeBits(int32(numCodegens-4), 4)
286
287     for i := 0; i < numCodegens; i++ {
288         value := w.codegenEncoding.codeBits[codegenC

```

```

289         w.writeBits(int32(value), 3)
290     }
291
292     i := 0
293     for {
294         var codeWord int = int(w.codegen[i])
295         i++
296         if codeWord == badCode {
297             break
298         }
299         // The low byte contains the actual code to
300         w.writeCode(w.codegenEncoding, uint32(codeWo
301
302         switch codeWord {
303         case 16:
304             w.writeBits(int32(w.codegen[i]), 2)
305             i++
306             break
307         case 17:
308             w.writeBits(int32(w.codegen[i]), 3)
309             i++
310             break
311         case 18:
312             w.writeBits(int32(w.codegen[i]), 7)
313             i++
314             break
315         }
316     }
317 }
318
319 func (w *huffmanBitWriter) writeStoredHeader(length int, isE
320     if w.err != nil {
321         return
322     }
323     var flag int32
324     if isEof {
325         flag = 1
326     }
327     w.writeBits(flag, 3)
328     w.flush()
329     w.writeBits(int32(length), 16)
330     w.writeBits(int32(^uint16(length)), 16)
331 }
332
333 func (w *huffmanBitWriter) writeFixedHeader(isEof bool) {
334     if w.err != nil {
335         return
336     }
337     // Indicate that we are a fixed Huffman block

```

```

338     var value int32 = 2
339     if isEof {
340         value = 3
341     }
342     w.writeBits(value, 3)
343 }
344
345 func (w *huffmanBitWriter) writeBlock(tokens []token, eof bo
346     if w.err != nil {
347         return
348     }
349     for i := range w.literalFreq {
350         w.literalFreq[i] = 0
351     }
352     for i := range w.offsetFreq {
353         w.offsetFreq[i] = 0
354     }
355
356     n := len(tokens)
357     tokens = tokens[0 : n+1]
358     tokens[n] = endBlockMarker
359
360     for _, t := range tokens {
361         switch t.typ() {
362         case literalType:
363             w.literalFreq[t.literal()]++
364         case matchType:
365             length := t.length()
366             offset := t.offset()
367             w.literalFreq[lengthCodesStart+length]++
368             w.offsetFreq[offsetCode(offset)]++
369         }
370     }
371
372     // get the number of literals
373     numLiterals := len(w.literalFreq)
374     for w.literalFreq[numLiterals-1] == 0 {
375         numLiterals--
376     }
377     // get the number of offsets
378     numOffsets := len(w.offsetFreq)
379     for numOffsets > 0 && w.offsetFreq[numOffsets-1] ==
380         numOffsets--
381     }
382     if numOffsets == 0 {
383         // We haven't found a single match. If we wa
384         // we should count at least one offset to be
385         w.offsetFreq[0] = 1
386         numOffsets = 1
387     }

```

```

388
389     w.literalEncoding.generate(w.literalFreq, 15)
390     w.offsetEncoding.generate(w.offsetFreq, 15)
391
392     storedBytes := 0
393     if input != nil {
394         storedBytes = len(input)
395     }
396     var extraBits int64
397     var storedSize int64 = math.MaxInt64
398     if storedBytes <= maxStoreBlockSize && input != nil
399         storedSize = int64((storedBytes + 5) * 8)
400         // We only bother calculating the costs of t
401         // the length of offset fields (which will b
402         // and dynamic encoding), if we need to comp
403         // against stored encoding.
404         for lengthCode := lengthCodesStart + 8; leng
405             // First eight length codes have ext
406             extraBits += int64(w.literalFreq[len
407         }
408         for offsetCode := 4; offsetCode < numOffsets
409             // First four offset codes have extr
410             extraBits += int64(w.offsetFreq[offs
411     }
412 }
413
414 // Figure out smallest code.
415 // Fixed Huffman baseline.
416 var size = int64(3) +
417     fixedLiteralEncoding.bitLength(w.literalFreq)
418     fixedOffsetEncoding.bitLength(w.offsetFreq)
419     extraBits
420 var literalEncoding = fixedLiteralEncoding
421 var offsetEncoding = fixedOffsetEncoding
422
423 // Dynamic Huffman?
424 var numCodegens int
425
426 // Generate codegen and codegenFrequencies, which in
427 // the literalEncoding and the offsetEncoding.
428 w.generateCodegen(numLiterals, numOffsets)
429 w.codegenEncoding.generate(w.codegenFreq, 7)
430 numCodegens = len(w.codegenFreq)
431 for numCodegens > 4 && w.codegenFreq[codegenOrder[nu
432     numCodegens--
433 }
434 dynamicHeader := int64(3+5+5+4+(3*numCodegens)) +
435     w.codegenEncoding.bitLength(w.codegenFreq) +
436     int64(extraBits) +

```

```

437         int64(w.codegenFreq[16]*2) +
438         int64(w.codegenFreq[17]*3) +
439         int64(w.codegenFreq[18]*7)
440     dynamicSize := dynamicHeader +
441         w.literalEncoding.bitLength(w.literalFreq) +
442         w.offsetEncoding.bitLength(w.offsetFreq)
443
444     if dynamicSize < size {
445         size = dynamicSize
446         literalEncoding = w.literalEncoding
447         offsetEncoding = w.offsetEncoding
448     }
449
450     // Stored bytes?
451     if storedSize < size {
452         w.writeStoredHeader(storedBytes, eof)
453         w.writeBytes(input[0:storedBytes])
454         return
455     }
456
457     // Huffman.
458     if literalEncoding == fixedLiteralEncoding {
459         w.writeFixedHeader(eof)
460     } else {
461         w.writeDynamicHeader(numLiterals, numOffsets)
462     }
463     for _, t := range tokens {
464         switch t.typ() {
465         case literalType:
466             w.writeCode(literalEncoding, t.liter
467             break
468         case matchType:
469             // Write the length
470             length := t.length()
471             lengthCode := lengthCode(length)
472             w.writeCode(literalEncoding, lengthC
473             extraLengthBits := int32(lengthExtra
474             if extraLengthBits > 0 {
475                 extraLength := int32(length
476                 w.writeBits(extraLength, ext
477             }
478             // Write the offset
479             offset := t.offset()
480             offsetCode := offsetCode(offset)
481             w.writeCode(offsetEncoding, offsetCo
482             extraOffsetBits := int32(offsetExtra
483             if extraOffsetBits > 0 {
484                 extraOffset := int32(offset
485                 w.writeBits(extraOffset, ext

```

```
486             }
487             break
488         default:
489             panic("unknown token type: " + strin
490         }
491     }
492 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/flate/huffman_code.

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package flate
6
7 import (
8     "math"
9     "sort"
10 )
11
12 type huffmanEncoder struct {
13     codeBits []uint8
14     code     []uint16
15 }
16
17 type literalNode struct {
18     literal uint16
19     freq    int32
20 }
21
22 type chain struct {
23     // The sum of the leaves in this tree
24     freq int32
25
26     // The number of literals to the left of this item a
27     leafCount int32
28
29     // The right child of this chain in the previous level
30     up *chain
31 }
32
33 type levelInfo struct {
34     // Our level. for better printing
35     level int32
36
37     // The most recent chain generated for this level
38     lastChain *chain
39
40     // The frequency of the next character to add to this
41     nextCharFreq int32
```

```

42
43     // The frequency of the next pair (from level below)
44     // Only valid if the "needed" value of the next lowe
45     nextPairFreq int32
46
47     // The number of chains remaining to generate for th
48     // up to the next level
49     needed int32
50
51     // The levelInfo for level+1
52     up *levelInfo
53
54     // The levelInfo for level-1
55     down *levelInfo
56 }
57
58 func maxNode() literalNode { return literalNode{math.MaxUint
59
60 func newHuffmanEncoder(size int) *huffmanEncoder {
61     return &huffmanEncoder{make([]uint8, size), make([]u
62 }
63
64 // Generates a HuffmanCode corresponding to the fixed litera
65 func generateFixedLiteralEncoding() *huffmanEncoder {
66     h := newHuffmanEncoder(maxLit)
67     codeBits := h.codeBits
68     code := h.code
69     var ch uint16
70     for ch = 0; ch < maxLit; ch++ {
71         var bits uint16
72         var size uint8
73         switch {
74         case ch < 144:
75             // size 8, 000110000 .. 10111111
76             bits = ch + 48
77             size = 8
78             break
79         case ch < 256:
80             // size 9, 110010000 .. 111111111
81             bits = ch + 400 - 144
82             size = 9
83             break
84         case ch < 280:
85             // size 7, 0000000 .. 0010111
86             bits = ch - 256
87             size = 7
88             break
89         default:
90             // size 8, 11000000 .. 11000111
91             bits = ch + 192 - 280

```

```

92             size = 8
93         }
94         codeBits[ch] = size
95         code[ch] = reverseBits(bits, size)
96     }
97     return h
98 }
99
100 func generateFixedOffsetEncoding() *huffmanEncoder {
101     h := newHuffmanEncoder(30)
102     codeBits := h.codeBits
103     code := h.code
104     for ch := uint16(0); ch < 30; ch++ {
105         codeBits[ch] = 5
106         code[ch] = reverseBits(ch, 5)
107     }
108     return h
109 }
110
111 var fixedLiteralEncoding *huffmanEncoder = generateFixedLite
112 var fixedOffsetEncoding *huffmanEncoder = generateFixedOffse
113
114 func (h *huffmanEncoder) bitLength(freq []int32) int64 {
115     var total int64
116     for i, f := range freq {
117         if f != 0 {
118             total += int64(f) * int64(h.codeBits
119         }
120     }
121     return total
122 }
123
124 // Return the number of literals assigned to each bit size i
125 //
126 // This method is only called when list.length >= 3
127 // The cases of 0, 1, and 2 literals are handled by special
128 //
129 // list An array of the literals with non-zero frequencies
130 //     and their associated frequencies. The array
131 //     frequency, and has as its last element a spec
132 //     MaxInt32
133 // maxBits The maximum number of bits that should be use
134 // return An integer array in which array[i] indicates
135 //     that should be encoded in i bits.
136 func (h *huffmanEncoder) bitCounts(list []literalNode, maxBi
137     n := int32(len(list))
138     list = list[0 : n+1]
139     list[n] = maxNode()
140

```

```

141 // The tree can't have greater depth than n - 1, no
142 // saves a little bit of work in some small cases
143 if maxBits > n-1 {
144     maxBits = n - 1
145 }
146
147 // Create information about each of the levels.
148 // A bogus "Level 0" whose sole purpose is so that
149 // level1.prev.needed==0. This makes level1.nextPai
150 // be a legitimate value that never gets chosen.
151 top := &levelInfo{needed: 0}
152 chain2 := &chain{list[1].freq, 2, new(chain)}
153 for level := int32(1); level <= maxBits; level++ {
154     // For every level, the first two items are
155     // We initialize the levels as if we had alr
156     top = &levelInfo{
157         level:         level,
158         lastChain:     chain2,
159         nextCharFreq: list[2].freq,
160         nextPairFreq: list[0].freq + list[1]
161         down:         top,
162     }
163     top.down.up = top
164     if level == 1 {
165         top.nextPairFreq = math.MaxInt32
166     }
167 }
168
169 // We need a total of 2*n - 2 items at top level and
170 top.needed = 2*n - 4
171
172 l := top
173 for {
174     if l.nextPairFreq == math.MaxInt32 && l.next
175         // We've run out of both leafs and p
176         // End all calculations for this lev
177         // To m sure we never come back to t
178         // set nextPairFreq impossibly large
179         l.lastChain = nil
180         l.needed = 0
181         l = l.up
182         l.nextPairFreq = math.MaxInt32
183         continue
184     }
185
186     prevFreq := l.lastChain.freq
187     if l.nextCharFreq < l.nextPairFreq {
188         // The next item on this row is a le
189         n := l.lastChain.leafCount + 1

```

```

190         l.lastChain = &chain{l.nextCharFreq,
191         l.nextCharFreq = list[n].freq
192     } else {
193         // The next item on this row is a pa
194         // nextPairFreq isn't valid until we
195         // more values in the level below
196         l.lastChain = &chain{l.nextPairFreq,
197         l.down.needed = 2
198     }
199
200     if l.needed--; l.needed == 0 {
201         // We've done everything we need to
202         // Continue calculating one level up
203         // of that level with the sum of the
204         // this level.
205         up := l.up
206         if up == nil {
207             // All done!
208             break
209         }
210         up.nextPairFreq = prevFreq + l.lastC
211         l = up
212     } else {
213         // If we stole from below, move down
214         // for l.down.needed > 0 {
215         l = l.down
216     }
217 }
218 }
219
220 // Somethings is wrong if at the end, the top level
221 // all of the leaves.
222 if top.lastChain.leafCount != n {
223     panic("top.lastChain.leafCount != n")
224 }
225
226 bitCount := make([]int32, maxBits+1)
227 bits := 1
228 for chain := top.lastChain; chain.up != nil; chain =
229     // chain.leafCount gives the number of liter
230     // bits to encode.
231     bitCount[bits] = chain.leafCount - chain.up.
232     bits++
233 }
234 return bitCount
235 }
236
237 // Look at the leaves and assign them a bit count and an enc
238 // in RFC 1951 3.2.2
239 func (h *huffmanEncoder) assignEncodingAndSize(bitCount []in

```

```

240     code := uint16(0)
241     for n, bits := range bitCount {
242         code <<= 1
243         if n == 0 || bits == 0 {
244             continue
245         }
246         // The literals list[len(list)-bits] .. list
247         // are encoded using "bits" bits, and get th
248         // code, code + 1, .... The code values are
249         // assigned in literal order (not frequency
250         chunk := list[len(list)-int(bits):]
251         sortByLiteral(chunk)
252         for _, node := range chunk {
253             h.codeBits[node.literal] = uint8(n)
254             h.code[node.literal] = reverseBits(c
255             code++
256         }
257         list = list[0 : len(list)-int(bits)]
258     }
259 }
260
261 // Update this Huffman Code object to be the minimum code fo
262 //
263 // freq An array of frequencies, in which frequency[i] give
264 // maxBits The maximum number of bits to use for any litera
265 func (h *huffmanEncoder) generate(freq []int32, maxBits int3
266     list := make([]literalNode, len(freq)+1)
267     // Number of non-zero literals
268     count := 0
269     // Set list to be the set of all non-zero literals a
270     for i, f := range freq {
271         if f != 0 {
272             list[count] = literalNode{uint16(i),
273             count++
274         } else {
275             h.codeBits[i] = 0
276         }
277     }
278     // If freq[] is shorter than codeBits[], fill rest o
279     h.codeBits = h.codeBits[0:len(freq)]
280     list = list[0:count]
281     if count <= 2 {
282         // Handle the small cases here, because they
283         // two or fewer literals, everything has bit
284         for i, node := range list {
285             // "list" is in order of increasing
286             h.codeBits[node.literal] = 1
287             h.code[node.literal] = uint16(i)
288         }

```

```

289         return
290     }
291     sortByFreq(list)
292
293     // Get the number of literals for each bit count
294     bitCount := h.bitCounts(list, maxBits)
295     // And do the assignment
296     h.assignEncodingAndSize(bitCount, list)
297 }
298
299 type literalNodeSorter struct {
300     a []literalNode
301     less func(i, j int) bool
302 }
303
304 func (s literalNodeSorter) Len() int { return len(s.a) }
305
306 func (s literalNodeSorter) Less(i, j int) bool {
307     return s.less(i, j)
308 }
309
310 func (s literalNodeSorter) Swap(i, j int) { s.a[i], s.a[j] =
311
312 func sortByFreq(a []literalNode) {
313     s := &literalNodeSorter{a, func(i, j int) bool {
314         if a[i].freq == a[j].freq {
315             return a[i].literal < a[j].literal
316         }
317         return a[i].freq < a[j].freq
318     }}
319     sort.Sort(s)
320 }
321
322 func sortByLiteral(a []literalNode) {
323     s := &literalNodeSorter{a, func(i, j int) bool { ret
324     sort.Sort(s)
325 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/flate/inflate.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package flate implements the DEFLATE compressed data form
6 // RFC 1951. The gzip and zlib packages implement access to
7 // formats.
8 package flate
9
10 import (
11     "bufio"
12     "io"
13     "strconv"
14 )
15
16 const (
17     maxCodeLen = 16 // max length of Huffman code
18     maxHist    = 32768 // max history required
19     maxLit     = 286
20     maxDist    = 32
21     numCodes   = 19 // number of codes in Huffman meta-c
22 )
23
24 // A CorruptInputError reports the presence of corrupt input
25 type CorruptInputError int64
26
27 func (e CorruptInputError) Error() string {
28     return "flate: corrupt input before offset " + strconv
29 }
30
31 // An InternalError reports an error in the flate code itself
32 type InternalError string
33
34 func (e InternalError) Error() string { return "flate: inter
35
36 // A ReadError reports an error encountered while reading in
37 type ReadError struct {
38     Offset int64 // byte offset where error occurred
39     Err    error // error returned by underlying Read
40 }
41
```

```

42 func (e *ReadError) Error() string {
43     return "flate: read error at offset " + strconv.FormatInt(int64(e.Offset), 10)
44 }
45
46 // A WriteError reports an error encountered while writing a block.
47 type WriteError struct {
48     Offset int64 // byte offset where error occurred
49     Err     error // error returned by underlying Write
50 }
51
52 func (e *WriteError) Error() string {
53     return "flate: write error at offset " + strconv.FormatInt(int64(e.Offset), 10)
54 }
55
56 // Huffman decoder is based on
57 // J. Brian Connell, "A Huffman-Shannon-Fano Code,"
58 // Proceedings of the IEEE, 61(7) (July 1973), pp 1046-1047.
59 type huffmanDecoder struct {
60     // min, max code length
61     min, max int
62
63     // limit[i] = largest code word of length i
64     // Given code v of length n,
65     // need more bits if v > limit[n].
66     limit [maxCodeLen + 1]int
67
68     // base[i] = smallest code word of length i - sequence number
69     base [maxCodeLen + 1]int
70
71     // codes[seq number] = output code.
72     // Given code v of length n, value is
73     // codes[v - base[n]].
74     codes []int
75 }
76
77 // Initialize Huffman decoding tables from array of code lengths
78 func (h *huffmanDecoder) init(bits []int) bool {
79     // Count number of codes of each length,
80     // compute min and max length.
81     var count [maxCodeLen + 1]int
82     var min, max int
83     for _, n := range bits {
84         if n == 0 {
85             continue
86         }
87         if min == 0 || n < min {
88             min = n
89         }
90         if n > max {
91             max = n

```

```

92         }
93         count[n]++
94     }
95     if max == 0 {
96         return false
97     }
98
99     h.min = min
100    h.max = max
101
102    // For each code range, compute
103    // nextcode (first code of that length),
104    // limit (last code of that length), and
105    // base (offset from first code to sequence number).
106    code := 0
107    seq := 0
108    var nextcode [maxCodeLen]int
109    for i := min; i <= max; i++ {
110        n := count[i]
111        nextcode[i] = code
112        h.base[i] = code - seq
113        code += n
114        seq += n
115        h.limit[i] = code - 1
116        code <<= 1
117    }
118
119    // Make array mapping sequence numbers to codes.
120    if len(h.codes) < len(bits) {
121        h.codes = make([]int, len(bits))
122    }
123    for i, n := range bits {
124        if n == 0 {
125            continue
126        }
127        code := nextcode[n]
128        nextcode[n]++
129        seq := code - h.base[n]
130        h.codes[seq] = i
131    }
132    return true
133 }
134
135 // Hard-coded Huffman tables for DEFLATE algorithm.
136 // See RFC 1951, section 3.2.6.
137 var fixedHuffmanDecoder = huffmanDecoder{
138     7, 9,
139     [maxCodeLen + 1]int{7: 23, 199, 511},
140     [maxCodeLen + 1]int{7: 0, 24, 224},

```

```

141     []int{
142         // length 7: 256-279
143         256, 257, 258, 259, 260, 261, 262,
144         263, 264, 265, 266, 267, 268, 269,
145         270, 271, 272, 273, 274, 275, 276,
146         277, 278, 279,
147
148         // length 8: 0-143
149         0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
150         12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
151         22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
152         32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
153         42, 43, 44, 45, 46, 47, 48, 49, 50, 51,
154         52, 53, 54, 55, 56, 57, 58, 59, 60, 61,
155         62, 63, 64, 65, 66, 67, 68, 69, 70, 71,
156         72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
157         82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
158         92, 93, 94, 95, 96, 97, 98, 99, 100,
159         101, 102, 103, 104, 105, 106, 107, 108,
160         109, 110, 111, 112, 113, 114, 115, 116,
161         117, 118, 119, 120, 121, 122, 123, 124,
162         125, 126, 127, 128, 129, 130, 131, 132,
163         133, 134, 135, 136, 137, 138, 139, 140,
164         141, 142, 143,
165
166         // length 8: 280-287
167         280, 281, 282, 283, 284, 285, 286, 287,
168
169         // length 9: 144-255
170         144, 145, 146, 147, 148, 149, 150, 151,
171         152, 153, 154, 155, 156, 157, 158, 159,
172         160, 161, 162, 163, 164, 165, 166, 167,
173         168, 169, 170, 171, 172, 173, 174, 175,
174         176, 177, 178, 179, 180, 181, 182, 183,
175         184, 185, 186, 187, 188, 189, 190, 191,
176         192, 193, 194, 195, 196, 197, 198, 199,
177         200, 201, 202, 203, 204, 205, 206, 207,
178         208, 209, 210, 211, 212, 213, 214, 215,
179         216, 217, 218, 219, 220, 221, 222, 223,
180         224, 225, 226, 227, 228, 229, 230, 231,
181         232, 233, 234, 235, 236, 237, 238, 239,
182         240, 241, 242, 243, 244, 245, 246, 247,
183         248, 249, 250, 251, 252, 253, 254, 255,
184     },
185 }
186
187 // The actual read interface needed by NewReader.
188 // If the passed in io.Reader does not also have ReadByte,
189 // the NewReader will introduce its own buffering.

```

```

190 type Reader interface {
191     io.Reader
192     ReadByte() (c byte, err error)
193 }
194
195 // Decompress state.
196 type decompressor struct {
197     // Input source.
198     r      Reader
199     roffset int64
200     woffset int64
201
202     // Input bits, in top of b.
203     b  uint32
204     nb uint
205
206     // Huffman decoders for literal/length, distance.
207     h1, h2 huffmanDecoder
208
209     // Length arrays used to define Huffman codes.
210     bits      [maxLit + maxDist]int
211     codebits  [numCodes]int
212
213     // Output history, buffer.
214     hist [maxHist]byte
215     hp   int // current output position in buffer
216     hw   int // have written hist[0:hw] already
217     hfull bool // buffer has filled at least once
218
219     // Temporary buffer (avoids repeated allocation).
220     buf [4]byte
221
222     // Next step in the decompression,
223     // and decompression state.
224     step      func(*decompressor)
225     final     bool
226     err       error
227     toRead    []byte
228     h1, hd    *huffmanDecoder
229     copyLen   int
230     copyDist  int
231 }
232
233 func (f *decompressor) nextBlock() {
234     if f.final {
235         if f.hw != f.hp {
236             f.flush((*decompressor).nextBlock)
237             return
238         }
239         f.err = io.EOF

```

```

240         return
241     }
242     for f.nb < 1+2 {
243         if f.err = f.moreBits(); f.err != nil {
244             return
245         }
246     }
247     f.final = f.b&1 == 1
248     f.b >>= 1
249     typ := f.b & 3
250     f.b >>= 2
251     f.nb -= 1 + 2
252     switch typ {
253     case 0:
254         f.dataBlock()
255     case 1:
256         // compressed, fixed Huffman tables
257         f.h1 = &fixedHuffmanDecoder
258         f.hd = nil
259         f.huffmanBlock()
260     case 2:
261         // compressed, dynamic Huffman tables
262         if f.err = f.readHuffman(); f.err != nil {
263             break
264         }
265         f.h1 = &f.h1
266         f.hd = &f.h2
267         f.huffmanBlock()
268     default:
269         // 3 is reserved.
270         f.err = CorruptInputError(f.roffset)
271     }
272 }
273
274 func (f *decompressor) Read(b []byte) (int, error) {
275     for {
276         if len(f.toRead) > 0 {
277             n := copy(b, f.toRead)
278             f.toRead = f.toRead[n:]
279             return n, nil
280         }
281         if f.err != nil {
282             return 0, f.err
283         }
284         f.step(f)
285     }
286     panic("unreachable")
287 }
288

```

```

289 func (f *decompressor) Close() error {
290     if f.err == io.EOF {
291         return nil
292     }
293     return f.err
294 }
295
296 // RFC 1951 section 3.2.7.
297 // Compression with dynamic Huffman codes
298
299 var codeOrder = [...]int{16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 1
300
301 func (f *decompressor) readHuffman() error {
302     // HLIT[5], HDIST[5], HCLLEN[4].
303     for f.nb < 5+5+4 {
304         if err := f.moreBits(); err != nil {
305             return err
306         }
307     }
308     nlit := int(f.b&0x1F) + 257
309     f.b >>= 5
310     ndist := int(f.b&0x1F) + 1
311     f.b >>= 5
312     nclen := int(f.b&0xF) + 4
313     f.b >>= 4
314     f.nb -= 5 + 5 + 4
315
316     // (HCLLEN+4)*3 bits: code lengths in the magic code0
317     for i := 0; i < nclen; i++ {
318         for f.nb < 3 {
319             if err := f.moreBits(); err != nil {
320                 return err
321             }
322         }
323         f.codebits[codeOrder[i]] = int(f.b & 0x7)
324         f.b >>= 3
325         f.nb -= 3
326     }
327     for i := nclen; i < len(codeOrder); i++ {
328         f.codebits[codeOrder[i]] = 0
329     }
330     if !f.h1.init(f.codebits[0:]) {
331         return CorruptInputError(f.roffset)
332     }
333
334     // HLIT + 257 code lengths, HDIST + 1 code lengths,
335     // using the code length Huffman code.
336     for i, n := 0, nlit+ndist; i < n; {
337         x, err := f.huffSym(&f.h1)

```

```

338         if err != nil {
339             return err
340         }
341         if x < 16 {
342             // Actual length.
343             f.bits[i] = x
344             i++
345             continue
346         }
347         // Repeat previous length or zero.
348         var rep int
349         var nb uint
350         var b int
351         switch x {
352         default:
353             return InternalError("unexpected len
354         case 16:
355             rep = 3
356             nb = 2
357             if i == 0 {
358                 return CorruptInputError(f.r
359             }
360             b = f.bits[i-1]
361         case 17:
362             rep = 3
363             nb = 3
364             b = 0
365         case 18:
366             rep = 11
367             nb = 7
368             b = 0
369         }
370         for f.nb < nb {
371             if err := f.moreBits(); err != nil {
372                 return err
373             }
374         }
375         rep += int(f.b & uint32(1<<nb-1))
376         f.b >>= nb
377         f.nb -= nb
378         if i+rep > n {
379             return CorruptInputError(f.roffset)
380         }
381         for j := 0; j < rep; j++ {
382             f.bits[i] = b
383             i++
384         }
385     }
386
387     if !f.h1.init(f.bits[0:nlit]) || !f.h2.init(f.bits[n

```

```

388         return CorruptInputError(f.roffset)
389     }
390
391     return nil
392 }
393
394 // Decode a single Huffman block from f.
395 // hl and hd are the Huffman states for the lit/length value
396 // and the distance values, respectively. If hd == nil, use
397 // fixed distance encoding associated with fixed Huffman block
398 func (f *decompressor) huffmanBlock() {
399     for {
400         v, err := f.huffSym(f.hl)
401         if err != nil {
402             f.err = err
403             return
404         }
405         var n uint // number of bits extra
406         var length int
407         switch {
408         case v < 256:
409             f.hist[f.hp] = byte(v)
410             f.hp++
411             if f.hp == len(f.hist) {
412                 // After the flush, continue
413                 f.flush((*decompressor).huff)
414                 return
415             }
416             continue
417         case v == 256:
418             // Done with Huffman block; read next
419             f.step = (*decompressor).nextBlock
420             return
421         // otherwise, reference to older data
422         case v < 265:
423             length = v - (257 - 3)
424             n = 0
425         case v < 269:
426             length = v*2 - (265*2 - 11)
427             n = 1
428         case v < 273:
429             length = v*4 - (269*4 - 19)
430             n = 2
431         case v < 277:
432             length = v*8 - (273*8 - 35)
433             n = 3
434         case v < 281:
435             length = v*16 - (277*16 - 67)
436             n = 4

```

```

437     case v < 285:
438         length = v*32 - (281*32 - 131)
439         n = 5
440     default:
441         length = 258
442         n = 0
443     }
444     if n > 0 {
445         for f.nb < n {
446             if err = f.moreBits(); err !
447                 f.err = err
448                 return
449         }
450     }
451     length += int(f.b & uint32(1<<n-1))
452     f.b >>= n
453     f.nb -= n
454 }
455
456 var dist int
457 if f.hd == nil {
458     for f.nb < 5 {
459         if err = f.moreBits(); err !
460             f.err = err
461             return
462     }
463 }
464 dist = int(reverseByte[(f.b&0x1F)<<3
465 f.b >>= 5
466 f.nb -= 5
467 } else {
468     if dist, err = f.huffSym(f.hd); err
469         f.err = err
470         return
471     }
472 }
473
474 switch {
475 case dist < 4:
476     dist++
477 case dist >= 30:
478     f.err = CorruptInputError(f.roffset)
479     return
480 default:
481     nb := uint(dist-2) >> 1
482     // have 1 bit in bottom of dist, need
483     extra := (dist & 1) << nb
484     for f.nb < nb {
485         if err = f.moreBits(); err !

```

```

486                                     f.err = err
487                                     return
488                                 }
489                             }
490                             extra |= int(f.b & uint32(1<<nb-1))
491                             f.b >>= nb
492                             f.nb -= nb
493                             dist = 1<<(nb+1) + 1 + extra
494                         }
495
496                         // Copy history[-dist:-dist+length] into out
497                         if dist > len(f.hist) {
498                             f.err = InternalError("bad history d
499                             return
500                         }
501
502                         // No check on length; encoding can be presc
503                         if !f.hfull && dist > f.hp {
504                             f.err = CorruptInputError(f.roffset)
505                             return
506                         }
507
508                         p := f.hp - dist
509                         if p < 0 {
510                             p += len(f.hist)
511                         }
512                         for i := 0; i < length; i++ {
513                             f.hist[f.hp] = f.hist[p]
514                             f.hp++
515                             p++
516                             if f.hp == len(f.hist) {
517                                 // After flush continue copy
518                                 f.copyLen = length - (i + 1)
519                                 f.copyDist = dist
520                                 f.flush((*decompressor).copy
521                                 return
522                             }
523                             if p == len(f.hist) {
524                                 p = 0
525                             }
526                         }
527                     }
528                 panic("unreached")
529             }
530
531             func (f *decompressor) copyHuff() {
532                 length := f.copyLen
533                 dist := f.copyDist
534                 p := f.hp - dist
535                 if p < 0 {

```

```

536         p += len(f.hist)
537     }
538     for i := 0; i < length; i++ {
539         f.hist[f.hp] = f.hist[p]
540         f.hp++
541         p++
542         if f.hp == len(f.hist) {
543             f.copyLen = length - (i + 1)
544             f.flush((*decompressor).copyHuff)
545             return
546         }
547         if p == len(f.hist) {
548             p = 0
549         }
550     }
551
552     // Continue processing Huffman block.
553     f.huffmanBlock()
554 }
555
556 // Copy a single uncompressed data block from input to output
557 func (f *decompressor) dataBlock() {
558     // Uncompressed.
559     // Discard current half-byte.
560     f.nb = 0
561     f.b = 0
562
563     // Length then ones-complement of length.
564     nr, err := io.ReadFull(f.r, f.buf[0:4])
565     f.roffset += int64(nr)
566     if err != nil {
567         f.err = &ReadError{f.roffset, err}
568         return
569     }
570     n := int(f.buf[0]) | int(f.buf[1])<<8
571     nn := int(f.buf[2]) | int(f.buf[3])<<8
572     if uint16(nn) != uint16(^n) {
573         f.err = CorruptInputError(f.roffset)
574         return
575     }
576
577     if n == 0 {
578         // 0-length block means sync
579         f.flush((*decompressor).nextBlock)
580         return
581     }
582
583     f.copyLen = n
584     f.copyData()

```

```

585 }
586
587 func (f *decompressor) copyData() {
588     // Read f.dataLen bytes into history,
589     // pausing for reads as history fills.
590     n := f.copyLen
591     for n > 0 {
592         m := len(f.hist) - f.hp
593         if m > n {
594             m = n
595         }
596         m, err := io.ReadFull(f.r, f.hist[f.hp:f.hp+m])
597         f.roffset += int64(m)
598         if err != nil {
599             f.err = &ReadError{f.roffset, err}
600             return
601         }
602         n -= m
603         f.hp += m
604         if f.hp == len(f.hist) {
605             f.copyLen = n
606             f.flush((*decompressor).copyData)
607             return
608         }
609     }
610     f.step = (*decompressor).nextBlock
611 }
612
613 func (f *decompressor) setDict(dict []byte) {
614     if len(dict) > len(f.hist) {
615         // Will only remember the tail.
616         dict = dict[len(dict)-len(f.hist):]
617     }
618
619     f.hp = copy(f.hist[:], dict)
620     if f.hp == len(f.hist) {
621         f.hp = 0
622         f.hfull = true
623     }
624     f.hw = f.hp
625 }
626
627 func (f *decompressor) moreBits() error {
628     c, err := f.r.ReadByte()
629     if err != nil {
630         if err == io.EOF {
631             err = io.ErrUnexpectedEOF
632         }
633         return err

```

```

634     }
635     f.roffset++
636     f.b |= uint32(c) << f.nb
637     f.nb += 8
638     return nil
639 }
640
641 // Read the next Huffman-encoded symbol from f according to
642 func (f *decompressor) huffSym(h *huffmanDecoder) (int, error) {
643     for n := uint(h.min); n <= uint(h.max); n++ {
644         lim := h.limit[n]
645         if lim == -1 {
646             continue
647         }
648         for f.nb < n {
649             if err := f.moreBits(); err != nil {
650                 return 0, err
651             }
652         }
653         v := int(f.b & uint32(1<<n-1))
654         v <<= 16 - n
655         v = int(reverseByte[v>>8]) | int(reverseByte
656             if v <= lim {
657                 f.b >>= n
658                 f.nb -= n
659                 return h.codes[v-h.base[n]], nil
660             }
661         }
662     }
663     return 0, CorruptInputError(f.roffset)
664 }
665 // Flush any buffered output to the underlying writer.
666 func (f *decompressor) flush(step func(*decompressor)) {
667     f.toRead = f.hist[f.hw:f.hp]
668     f.woffset += int64(f.hp - f.hw)
669     f.hw = f.hp
670     if f.hp == len(f.hist) {
671         f.hp = 0
672         f.hw = 0
673         f.hfull = true
674     }
675     f.step = step
676 }
677
678 func makeReader(r io.Reader) Reader {
679     if rr, ok := r.(Reader); ok {
680         return rr
681     }
682     return bufio.NewReader(r)
683 }

```

```

684
685 // NewReader returns a new ReadCloser that can be used
686 // to read the uncompressed version of r. It is the caller'
687 // responsibility to call Close on the ReadCloser when
688 // finished reading.
689 func NewReader(r io.Reader) io.ReadCloser {
690     var f decompressor
691     f.r = makeReader(r)
692     f.step = (*decompressor).nextBlock
693     return &f
694 }
695
696 // NewReaderDict is like NewReader but initializes the reader
697 // with a preset dictionary. The returned Reader behaves as
698 // the uncompressed data stream started with the given dictionary
699 // which has already been read. NewReaderDict is typically
700 // used to read data compressed by NewWriterDict.
701 func NewReaderDict(r io.Reader, dict []byte) io.ReadCloser {
702     var f decompressor
703     f.setDict(dict)
704     f.r = makeReader(r)
705     f.step = (*decompressor).nextBlock
706     return &f
707 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/flate/reverse_bits.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package flate
6
7 var reverseByte = [256]byte{
8     0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0,
9     0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0,
10    0x08, 0x88, 0x48, 0xc8, 0x28, 0xa8, 0x68, 0xe8,
11    0x18, 0x98, 0x58, 0xd8, 0x38, 0xb8, 0x78, 0xf8,
12    0x04, 0x84, 0x44, 0xc4, 0x24, 0xa4, 0x64, 0xe4,
13    0x14, 0x94, 0x54, 0xd4, 0x34, 0xb4, 0x74, 0xf4,
14    0x0c, 0x8c, 0x4c, 0xcc, 0x2c, 0xac, 0x6c, 0xec,
15    0x1c, 0x9c, 0x5c, 0xdc, 0x3c, 0xbc, 0x7c, 0xfc,
16    0x02, 0x82, 0x42, 0xc2, 0x22, 0xa2, 0x62, 0xe2,
17    0x12, 0x92, 0x52, 0xd2, 0x32, 0xb2, 0x72, 0xf2,
18    0x0a, 0x8a, 0x4a, 0xca, 0x2a, 0xaa, 0x6a, 0xea,
19    0x1a, 0x9a, 0x5a, 0xda, 0x3a, 0xba, 0x7a, 0xfa,
20    0x06, 0x86, 0x46, 0xc6, 0x26, 0xa6, 0x66, 0xe6,
21    0x16, 0x96, 0x56, 0xd6, 0x36, 0xb6, 0x76, 0xf6,
22    0x0e, 0x8e, 0x4e, 0xce, 0x2e, 0xae, 0x6e, 0xee,
23    0x1e, 0x9e, 0x5e, 0xde, 0x3e, 0xbe, 0x7e, 0xfe,
24    0x01, 0x81, 0x41, 0xc1, 0x21, 0xa1, 0x61, 0xe1,
25    0x11, 0x91, 0x51, 0xd1, 0x31, 0xb1, 0x71, 0xf1,
26    0x09, 0x89, 0x49, 0xc9, 0x29, 0xa9, 0x69, 0xe9,
27    0x19, 0x99, 0x59, 0xd9, 0x39, 0xb9, 0x79, 0xf9,
28    0x05, 0x85, 0x45, 0xc5, 0x25, 0xa5, 0x65, 0xe5,
29    0x15, 0x95, 0x55, 0xd5, 0x35, 0xb5, 0x75, 0xf5,
30    0x0d, 0x8d, 0x4d, 0xcd, 0x2d, 0xad, 0x6d, 0xed,
31    0x1d, 0x9d, 0x5d, 0xdd, 0x3d, 0xbd, 0x7d, 0xfd,
32    0x03, 0x83, 0x43, 0xc3, 0x23, 0xa3, 0x63, 0xe3,
33    0x13, 0x93, 0x53, 0xd3, 0x33, 0xb3, 0x73, 0xf3,
34    0x0b, 0x8b, 0x4b, 0xcb, 0x2b, 0xab, 0x6b, 0xeb,
35    0x1b, 0x9b, 0x5b, 0xdb, 0x3b, 0xbb, 0x7b, 0xfb,
36    0x07, 0x87, 0x47, 0xc7, 0x27, 0xa7, 0x67, 0xe7,
37    0x17, 0x97, 0x57, 0xd7, 0x37, 0xb7, 0x77, 0xf7,
38    0x0f, 0x8f, 0x4f, 0xcf, 0x2f, 0xaf, 0x6f, 0xef,
39    0x1f, 0x9f, 0x5f, 0xdf, 0x3f, 0xbf, 0x7f, 0xff,
40 }
41
```

```
42 func reverseUint16(v uint16) uint16 {
43     return uint16(reverseByte[v>>8]) | uint16(reverseByt
44 }
45
46 func reverseBits(number uint16, bitLength byte) uint16 {
47     return reverseUint16(number << uint8(16-bitLength))
48 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/flate/token.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package flate
6
7 const (
8     // 2 bits:  type  0 = literal  1=EOF  2=Match  3=
9     // 8 bits:  xlength = length - MIN_MATCH_LENGTH
10    // 22 bits  xoffset = offset - MIN_OFFSET_SIZE, or
11    lengthShift = 22
12    offsetMask  = 1<<lengthShift - 1
13    typeMask    = 3 << 30
14    literalType = 0 << 30
15    matchType   = 1 << 30
16 )
17
18 // The length code for length X (MIN_MATCH_LENGTH <= X <= MA
19 // is lengthCodes[length - MIN_MATCH_LENGTH]
20 var lengthCodes = [...]uint32{
21     0, 1, 2, 3, 4, 5, 6, 7, 8, 8,
22     9, 9, 10, 10, 11, 11, 12, 12, 12, 12,
23     13, 13, 13, 13, 14, 14, 14, 14, 15, 15,
24     15, 15, 16, 16, 16, 16, 16, 16, 16, 16,
25     17, 17, 17, 17, 17, 17, 17, 17, 18, 18,
26     18, 18, 18, 18, 18, 18, 19, 19, 19, 19,
27     19, 19, 19, 19, 20, 20, 20, 20, 20, 20,
28     20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
29     21, 21, 21, 21, 21, 21, 21, 21, 21, 21,
30     21, 21, 21, 21, 21, 21, 22, 22, 22, 22,
31     22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
32     22, 22, 23, 23, 23, 23, 23, 23, 23, 23,
33     23, 23, 23, 23, 23, 23, 23, 23, 24, 24,
34     24, 24, 24, 24, 24, 24, 24, 24, 24, 24,
35     24, 24, 24, 24, 24, 24, 24, 24, 24, 24,
36     24, 24, 24, 24, 24, 24, 24, 24, 24, 24,
37     25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
38     25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
39     25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
40     25, 25, 26, 26, 26, 26, 26, 26, 26, 26,
41     26, 26, 26, 26, 26, 26, 26, 26, 26, 26,
```

```

42         26, 26, 26, 26, 26, 26, 26, 26, 26, 26,
43         26, 26, 26, 26, 27, 27, 27, 27, 27, 27,
44         27, 27, 27, 27, 27, 27, 27, 27, 27, 27,
45         27, 27, 27, 27, 27, 27, 27, 27, 27, 27,
46         27, 27, 27, 27, 27, 28,
47     }
48
49     var offsetCodes = [...]uint32{
50         0, 1, 2, 3, 4, 4, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7,
51         8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9,
52         10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
53         11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11,
54         12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
55         12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
56         13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
57         13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
58         14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14,
59         14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14,
60         14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14,
61         14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14,
62         15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
63         15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
64         15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
65         15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
66     }
67
68     type token uint32
69
70     // Convert a literal into a literal token.
71     func literalToken(literal uint32) token { return token(liter
72
73     // Convert a < xlength, xoffset > pair into a match token.
74     func matchToken(xlength uint32, xoffset uint32) token {
75         return token(matchType + xlength<<lengthShift + xoff
76     }
77
78     // Returns the type of a token
79     func (t token) typ() uint32 { return uint32(t) & typeMask }
80
81     // Returns the literal of a literal token
82     func (t token) literal() uint32 { return uint32(t - literalT
83
84     // Returns the extra offset of a match token
85     func (t token) offset() uint32 { return uint32(t) & offsetMa
86
87     func (t token) length() uint32 { return uint32((t - matchTyp
88
89     func lengthCode(len uint32) uint32 { return lengthCodes[len]
90
91     // Returns the offset code corresponding to a specific offse

```

```
92 func offsetCode(off uint32) uint32 {
93     const n = uint32(len(offsetCodes))
94     switch {
95     case off < n:
96         return offsetCodes[off]
97     case off>>7 < n:
98         return offsetCodes[off>>7] + 14
99     default:
100         return offsetCodes[off>>14] + 28
101     }
102     panic("unreachable")
103 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/gzip/gunzip.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package gzip implements reading and writing of gzip forma
6 // as specified in RFC 1952.
7 package gzip
8
9 import (
10     "bufio"
11     "compress/flate"
12     "errors"
13     "hash"
14     "hash/crc32"
15     "io"
16     "time"
17 )
18
19 const (
20     gzipID1      = 0x1f
21     gzipID2      = 0x8b
22     gzipDeflate  = 8
23     flagText     = 1 << 0
24     flagHdrCrc   = 1 << 1
25     flagExtra    = 1 << 2
26     flagName     = 1 << 3
27     flagComment  = 1 << 4
28 )
29
30 func makeReader(r io.Reader) flate.Reader {
31     if rr, ok := r.(flate.Reader); ok {
32         return rr
33     }
34     return bufio.NewReader(r)
35 }
36
37 var (
38     // ErrChecksum is returned when reading GZIP data th
39     ErrChecksum = errors.New("gzip: invalid checksum")
40     // ErrHeader is returned when reading GZIP data that
41     ErrHeader = errors.New("gzip: invalid header")

```

```

42 )
43
44 // The gzip file stores a header giving metadata about the c
45 // That header is exposed as the fields of the Writer and Re
46 type Header struct {
47     Comment string // comment
48     Extra []byte // "extra data"
49     ModTime time.Time // modification time
50     Name string // file name
51     OS byte // operating system type
52 }
53
54 // A Reader is an io.Reader that can be read to retrieve
55 // uncompressed data from a gzip-format compressed file.
56 //
57 // In general, a gzip file can be a concatenation of gzip fi
58 // each with its own header. Reads from the Reader
59 // return the concatenation of the uncompressed data of each
60 // Only the first header is recorded in the Reader fields.
61 //
62 // Gzip files store a length and checksum of the uncompress
63 // The Reader will return a ErrChecksum when Read
64 // reaches the end of the uncompressed data if it does not
65 // have the expected length or checksum. Clients should tre
66 // returned by Read as tentative until they receive the io.E
67 // marking the end of the data.
68 type Reader struct {
69     Header
70     r flate.Reader
71     decompressor io.ReadCloser
72     digest hash.Hash32
73     size uint32
74     flg byte
75     buf [512]byte
76     err error
77 }
78
79 // NewReader creates a new Reader reading the given reader.
80 // The implementation buffers input and may read more data t
81 // It is the caller's responsibility to call Close on the Re
82 func NewReader(r io.Reader) (*Reader, error) {
83     z := new(Reader)
84     z.r = makeReader(r)
85     z.digest = crc32.NewIEEE()
86     if err := z.readHeader(true); err != nil {
87         return nil, err
88     }
89     return z, nil
90 }
91

```

```

92 // GZIP (RFC 1952) is little-endian, unlike ZLIB (RFC 1950).
93 func get4(p []byte) uint32 {
94     return uint32(p[0]) | uint32(p[1])<<8 | uint32(p[2])
95 }
96
97 func (z *Reader) readString() (string, error) {
98     var err error
99     needconv := false
100    for i := 0; ; i++ {
101        if i >= len(z.buf) {
102            return "", ErrHeader
103        }
104        z.buf[i], err = z.r.ReadByte()
105        if err != nil {
106            return "", err
107        }
108        if z.buf[i] > 0x7f {
109            needconv = true
110        }
111        if z.buf[i] == 0 {
112            // GZIP (RFC 1952) specifies that st
113            if needconv {
114                s := make([]rune, 0, i)
115                for _, v := range z.buf[0:i]
116                    s = append(s, rune(v)
117            }
118            return string(s), nil
119        }
120        return string(z.buf[0:i]), nil
121    }
122 }
123 panic("not reached")
124 }
125
126 func (z *Reader) read2() (uint32, error) {
127     _, err := io.ReadFull(z.r, z.buf[0:2])
128     if err != nil {
129         return 0, err
130     }
131     return uint32(z.buf[0]) | uint32(z.buf[1])<<8, nil
132 }
133
134 func (z *Reader) readHeader(save bool) error {
135     _, err := io.ReadFull(z.r, z.buf[0:10])
136     if err != nil {
137         return err
138     }
139     if z.buf[0] != gzipID1 || z.buf[1] != gzipID2 || z.b
140     return ErrHeader

```

```

141     }
142     z.flg = z.buf[3]
143     if save {
144         z.ModTime = time.Unix(int64(get4(z.buf[4:8]))
145             // z.buf[8] is xfl, ignored
146         z.OS = z.buf[9]
147     }
148     z.digest.Reset()
149     z.digest.Write(z.buf[0:10])
150
151     if z.flg&flagExtra != 0 {
152         n, err := z.read2()
153         if err != nil {
154             return err
155         }
156         data := make([]byte, n)
157         if _, err = io.ReadFull(z.r, data); err != n
158             return err
159     }
160     if save {
161         z.Extra = data
162     }
163 }
164
165     var s string
166     if z.flg&flagName != 0 {
167         if s, err = z.readString(); err != nil {
168             return err
169         }
170         if save {
171             z.Name = s
172         }
173     }
174
175     if z.flg&flagComment != 0 {
176         if s, err = z.readString(); err != nil {
177             return err
178         }
179         if save {
180             z.Comment = s
181         }
182     }
183
184     if z.flg&flagHdrCrc != 0 {
185         n, err := z.read2()
186         if err != nil {
187             return err
188         }
189         sum := z.digest.Sum32() & 0xFFFF

```

```

190         if n != sum {
191             return ErrHeader
192         }
193     }
194
195     z.digest.Reset()
196     z.decompressor = flate.NewReader(z.r)
197     return nil
198 }
199
200 func (z *Reader) Read(p []byte) (n int, err error) {
201     if z.err != nil {
202         return 0, z.err
203     }
204     if len(p) == 0 {
205         return 0, nil
206     }
207
208     n, err = z.decompressor.Read(p)
209     z.digest.Write(p[0:n])
210     z.size += uint32(n)
211     if n != 0 || err != io.EOF {
212         z.err = err
213         return
214     }
215
216     // Finished file; check checksum + size.
217     if _, err := io.ReadFull(z.r, z.buf[0:8]); err != ni
218         z.err = err
219         return 0, err
220     }
221     crc32, isize := get4(z.buf[0:4]), get4(z.buf[4:8])
222     sum := z.digest.Sum32()
223     if sum != crc32 || isize != z.size {
224         z.err = ErrChecksum
225         return 0, z.err
226     }
227
228     // File is ok; is there another?
229     if err = z.readHeader(false); err != nil {
230         z.err = err
231         return
232     }
233
234     // Yes. Reset and read from it.
235     z.digest.Reset()
236     z.size = 0
237     return z.Read(p)
238 }
239

```

```
240 // Close closes the Reader. It does not close the underlying
241 func (z *Reader) Close() error { return z.decompressor.Close
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/gzip/gzip.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package gzip
6
7 import (
8     "compress/flate"
9     "errors"
10    "fmt"
11    "hash"
12    "hash/crc32"
13    "io"
14 )
15
16 // These constants are copied from the flate package, so tha
17 // "compress/gzip" does not also have to import "compress/fl
18 const (
19     NoCompression      = flate.NoCompression
20     BestSpeed           = flate.BestSpeed
21     BestCompression    = flate.BestCompression
22     DefaultCompression = flate.DefaultCompression
23 )
24
25 // A Writer is an io.WriteCloser that satisfies writes by co
26 // to its wrapped io.Writer.
27 type Writer struct {
28     Header
29     w      io.Writer
30     level  int
31     compressor io.WriteCloser
32     digest  hash.Hash32
33     size    uint32
34     closed  bool
35     buf     [10]byte
36     err     error
37 }
38
39 // NewWriter creates a new Writer that satisfies writes by c
40 // written to w.
41 //
```

```

42 // It is the caller's responsibility to call Close on the Wr
43 // Writes may be buffered and not flushed until Close.
44 //
45 // Callers that wish to set the fields in Writer.Header must
46 // the first call to Write or Close. The Comment and Name he
47 // UTF-8 strings in Go, but the underlying format requires N
48 // 8859-1 (Latin-1). NUL or non-Latin-1 runes in those strin
49 // error on Write.
50 func NewWriter(w io.Writer) *Writer {
51     z, _ := NewWriterLevel(w, DefaultCompression)
52     return z
53 }
54
55 // NewWriterLevel is like NewWriter but specifies the compre
56 // of assuming DefaultCompression.
57 //
58 // The compression level can be DefaultCompression, NoCompre
59 // integer value between BestSpeed and BestCompression inclu
60 // returned will be nil if the level is valid.
61 func NewWriterLevel(w io.Writer, level int) (*Writer, error)
62     if level < DefaultCompression || level > BestCompres
63         return nil, fmt.Errorf("gzip: invalid compre
64     }
65     return &Writer{
66         Header: Header{
67             OS: 255, // unknown
68         },
69         w:      w,
70         level: level,
71         digest: crc32.NewIEEE(),
72     }, nil
73 }
74
75 // GZIP (RFC 1952) is little-endian, unlike ZLIB (RFC 1950).
76 func put2(p []byte, v uint16) {
77     p[0] = uint8(v >> 0)
78     p[1] = uint8(v >> 8)
79 }
80
81 func put4(p []byte, v uint32) {
82     p[0] = uint8(v >> 0)
83     p[1] = uint8(v >> 8)
84     p[2] = uint8(v >> 16)
85     p[3] = uint8(v >> 24)
86 }
87
88 // writeBytes writes a length-prefixed byte slice to z.w.
89 func (z *Writer) writeBytes(b []byte) error {
90     if len(b) > 0xffff {
91         return errors.New("gzip.Write: Extra data is

```

```

92     }
93     put2(z.buf[0:2], uint16(len(b)))
94     _, err := z.w.Write(z.buf[0:2])
95     if err != nil {
96         return err
97     }
98     _, err = z.w.Write(b)
99     return err
100 }
101
102 // writeString writes a UTF-8 string s in GZIP's format to z
103 // GZIP (RFC 1952) specifies that strings are NUL-terminated
104 func (z *Writer) writeString(s string) (err error) {
105     // GZIP stores Latin-1 strings; error if non-Latin-1
106     needconv := false
107     for _, v := range s {
108         if v == 0 || v > 0xff {
109             return errors.New("gzip.Write: non-L
110         }
111         if v > 0x7f {
112             needconv = true
113         }
114     }
115     if needconv {
116         b := make([]byte, 0, len(s))
117         for _, v := range s {
118             b = append(b, byte(v))
119         }
120         _, err = z.w.Write(b)
121     } else {
122         _, err = io.WriteString(z.w, s)
123     }
124     if err != nil {
125         return err
126     }
127     // GZIP strings are NUL-terminated.
128     z.buf[0] = 0
129     _, err = z.w.Write(z.buf[0:1])
130     return err
131 }
132
133 // Write writes a compressed form of p to the underlying io.
134 // compressed bytes are not necessarily flushed until the Wr
135 func (z *Writer) Write(p []byte) (int, error) {
136     if z.err != nil {
137         return 0, z.err
138     }
139     var n int
140     // Write the GZIP header lazily.

```

```

141     if z.compressor == nil {
142         z.buf[0] = gzipID1
143         z.buf[1] = gzipID2
144         z.buf[2] = gzipDeflate
145         z.buf[3] = 0
146         if z.Extra != nil {
147             z.buf[3] |= 0x04
148         }
149         if z.Name != "" {
150             z.buf[3] |= 0x08
151         }
152         if z.Comment != "" {
153             z.buf[3] |= 0x10
154         }
155         put4(z.buf[4:8], uint32(z.ModTime.Unix()))
156         if z.level == BestCompression {
157             z.buf[8] = 2
158         } else if z.level == BestSpeed {
159             z.buf[8] = 4
160         } else {
161             z.buf[8] = 0
162         }
163         z.buf[9] = z.OS
164         n, z.err = z.w.Write(z.buf[0:10])
165         if z.err != nil {
166             return n, z.err
167         }
168         if z.Extra != nil {
169             z.err = z.writeBytes(z.Extra)
170             if z.err != nil {
171                 return n, z.err
172             }
173         }
174         if z.Name != "" {
175             z.err = z.writeString(z.Name)
176             if z.err != nil {
177                 return n, z.err
178             }
179         }
180         if z.Comment != "" {
181             z.err = z.writeString(z.Comment)
182             if z.err != nil {
183                 return n, z.err
184             }
185         }
186         z.compressor, _ = flate.NewWriter(z.w, z.lev
187     }
188     z.size += uint32(len(p))
189     z.digest.Write(p)

```

```

190         n, z.err = z.compressor.Write(p)
191         return n, z.err
192     }
193
194     // Close closes the Writer. It does not close the underlying
195     func (z *Writer) Close() error {
196         if z.err != nil {
197             return z.err
198         }
199         if z.closed {
200             return nil
201         }
202         z.closed = true
203         if z.compressor == nil {
204             z.Write(nil)
205             if z.err != nil {
206                 return z.err
207             }
208         }
209         z.err = z.compressor.Close()
210         if z.err != nil {
211             return z.err
212         }
213         put4(z.buf[0:4], z.digest.Sum32())
214         put4(z.buf[4:8], z.size)
215         _, z.err = z.w.Write(z.buf[0:8])
216         return z.err
217     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/lzw/reader.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package lzw implements the Lempel-Ziv-Welch compressed da
6 // described in T. A. Welch, ``A Technique for High-Performa
7 // Compression'', Computer, 17(6) (June 1984), pp 8-19.
8 //
9 // In particular, it implements LZW as used by the GIF, TIFF
10 // formats, which means variable-width codes up to 12 bits a
11 // two non-literal codes are a clear code and an EOF code.
12 package lzw
13
14 // TODO(nigeltao): check that TIFF and PDF use LZW in the sa
15 // modulo LSB/MSB packing order.
16
17 import (
18     "bufio"
19     "errors"
20     "fmt"
21     "io"
22 )
23
24 // Order specifies the bit ordering in an LZW data stream.
25 type Order int
26
27 const (
28     // LSB means Least Significant Bits first, as used i
29     LSB Order = iota
30     // MSB means Most Significant Bits first, as used in
31     // file formats.
32     MSB
33 )
34
35 const (
36     maxWidth          = 12
37     decoderInvalidCode = 0xffff
38     flushBuffer       = 1 << maxWidth
39 )
40
41 // decoder is the state from which the readXxx method conver
```

```

42 // stream into a code stream.
43 type decoder struct {
44     r      io.ByteReader
45     bits   uint32
46     nBits  uint
47     width  uint
48     read   func(*decoder) (uint16, error) // readLSB of
49     litWidth int                          // width in
50     err    error
51
52     // The first 1<<litWidth codes are literal codes.
53     // The next two codes mean clear and EOF.
54     // Other valid codes are in the range [lo, hi] where
55     // with the upper bound incrementing on each code seen
56     // overflow is the code at which hi overflows the code
57     // last is the most recently seen code, or decoderIn
58     clear, eof, hi, overflow, last uint16
59
60     // Each code c in [lo, hi] expands to two or more bytes
61     // suffix[c] is the last of these bytes.
62     // prefix[c] is the code for all but the last byte
63     // This code can either be a literal code or another
64     // The c == hi case is a special case.
65     suffix [1 << maxWidth]uint8
66     prefix [1 << maxWidth]uint16
67
68     // output is the temporary output buffer.
69     // Literal codes are accumulated from the start of the
70     // Non-literal codes decode to a sequence of suffixes
71     // written right-to-left from the end of the buffer
72     // to the start of the buffer.
73     // It is flushed when it contains >= 1<<maxWidth bytes
74     // so that there is always room to decode an entire
75     // output [2 * 1 << maxWidth]bytes
76     o      int // write index into output
77     toRead []byte // bytes to return from Read
78 }
79
80 // readLSB returns the next code for "Least Significant Bits
81 func (d *decoder) readLSB() (uint16, error) {
82     for d.nBits < d.width {
83         x, err := d.r.ReadByte()
84         if err != nil {
85             return 0, err
86         }
87         d.bits |= uint32(x) << d.nBits
88         d.nBits += 8
89     }
90     code := uint16(d.bits & (1<<d.width - 1))
91     d.bits >>= d.width

```

```

92         d.nBits -= d.width
93         return code, nil
94     }
95
96     // readMSB returns the next code for "Most Significant Bits
97     func (d *decoder) readMSB() (uint16, error) {
98         for d.nBits < d.width {
99             x, err := d.r.ReadByte()
100            if err != nil {
101                return 0, err
102            }
103            d.bits |= uint32(x) << (24 - d.nBits)
104            d.nBits += 8
105        }
106        code := uint16(d.bits >> (32 - d.width))
107        d.bits <<= d.width
108        d.nBits -= d.width
109        return code, nil
110    }
111
112    func (d *decoder) Read(b []byte) (int, error) {
113        for {
114            if len(d.toRead) > 0 {
115                n := copy(b, d.toRead)
116                d.toRead = d.toRead[n:]
117                return n, nil
118            }
119            if d.err != nil {
120                return 0, d.err
121            }
122            d.decode()
123        }
124        panic("unreachable")
125    }
126
127    // decode decompresses bytes from r and leaves them in d.toR
128    // read specifies how to decode bytes into codes.
129    // litWidth is the width in bits of literal codes.
130    func (d *decoder) decode() {
131        // Loop over the code stream, converting codes into
132        for {
133            code, err := d.read(d)
134            if err != nil {
135                if err == io.EOF {
136                    err = io.ErrUnexpectedEOF
137                }
138                d.err = err
139                return
140            }

```

```

141     switch {
142     case code < d.clear:
143         // We have a literal code.
144         d.output[d.o] = uint8(code)
145         d.o++
146         if d.last != decoderInvalidCode {
147             // Save what the hi code exp
148             d.suffix[d.hi] = uint8(code)
149             d.prefix[d.hi] = d.last
150         }
151     case code == d.clear:
152         d.width = 1 + uint(d.litWidth)
153         d.hi = d.eof
154         d.overflow = 1 << d.width
155         d.last = decoderInvalidCode
156         continue
157     case code == d.eof:
158         d.flush()
159         d.err = io.EOF
160         return
161     case code <= d.hi:
162         c, i := code, len(d.output)-1
163         if code == d.hi {
164             // code == hi is a special c
165             // followed by the head of t
166             // the prefix chain until we
167             c = d.last
168             for c >= d.clear {
169                 c = d.prefix[c]
170             }
171             d.output[i] = uint8(c)
172             i--
173             c = d.last
174         }
175         // Copy the suffix chain into output
176         for c >= d.clear {
177             d.output[i] = d.suffix[c]
178             i--
179             c = d.prefix[c]
180         }
181         d.output[i] = uint8(c)
182         d.o += copy(d.output[d.o:], d.output)
183         if d.last != decoderInvalidCode {
184             // Save what the hi code exp
185             d.suffix[d.hi] = uint8(c)
186             d.prefix[d.hi] = d.last
187         }
188     default:
189         d.err = errors.New("lzw: invalid cod

```

```

190             return
191         }
192         d.last, d.hi = code, d.hi+1
193         if d.hi >= d.overflow {
194             if d.width == maxWidth {
195                 d.last = decoderInvalidCode
196             } else {
197                 d.width++
198                 d.overflow <<= 1
199             }
200         }
201         if d.o >= flushBuffer {
202             d.flush()
203             return
204         }
205     }
206     panic("unreachable")
207 }
208
209 func (d *decoder) flush() {
210     d.toRead = d.output[:d.o]
211     d.o = 0
212 }
213
214 var errClosed = errors.New("compress/lzw: reader/writer is c
215
216 func (d *decoder) Close() error {
217     d.err = errClosed // in case any Reads come along
218     return nil
219 }
220
221 // NewReader creates a new io.ReadCloser that satisfies read
222 // the data read from r.
223 // It is the caller's responsibility to call Close on the Re
224 // finished reading.
225 // The number of bits to use for literal codes, litWidth, mu
226 // range [2,8] and is typically 8.
227 func NewReader(r io.Reader, order Order, litWidth int) io.Re
228     d := new(decoder)
229     switch order {
230     case LSB:
231         d.read = (*decoder).readLSB
232     case MSB:
233         d.read = (*decoder).readMSB
234     default:
235         d.err = errors.New("lzw: unknown order")
236         return d
237     }
238     if litWidth < 2 || 8 < litWidth {
239         d.err = fmt.Errorf("lzw: litWidth %d out of

```

```
240         return d
241     }
242     if br, ok := r.(io.ByteReader); ok {
243         d.r = br
244     } else {
245         d.r = bufio.NewReader(r)
246     }
247     d.litWidth = litWidth
248     d.width = 1 + uint(litWidth)
249     d.clear = uint16(1) << uint(litWidth)
250     d.eof, d.hi = d.clear+1, d.clear+1
251     d.overflow = uint16(1) << d.width
252     d.last = decoderInvalidCode
253
254     return d
255 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/lzw/writer.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package lzw
6
7 import (
8     "bufio"
9     "errors"
10    "fmt"
11    "io"
12 )
13
14 // A writer is a buffered, flushable writer.
15 type writer interface {
16     WriteByte(byte) error
17     Flush() error
18 }
19
20 // An errWriteCloser is an io.WriteCloser that always return
21 type errWriteCloser struct {
22     err error
23 }
24
25 func (e *errWriteCloser) Write([]byte) (int, error) {
26     return 0, e.err
27 }
28
29 func (e *errWriteCloser) Close() error {
30     return e.err
31 }
32
33 const (
34     // A code is a 12 bit value, stored as a uint32 when
35     // type conversions when shifting bits.
36     maxCode      = 1<<12 - 1
37     invalidCode = 1<<32 - 1
38     // There are 1<<12 possible codes, which is an upper
39     // valid hash table entries at any given point in ti
40     tableSize = 4 * 1 << 12
41     tableMask = tableSize - 1
```

```

42         // A hash table entry is a uint32. Zero is an invalid
43         // lower 12 bits of a valid entry must be a non-lite
44         invalidEntry = 0
45     )
46
47     // encoder is LZW compressor.
48     type encoder struct {
49         // w is the writer that compressed bytes are written
50         w writer
51         // order, write, bits, nBits and width are the state
52         // converting a code stream into a byte stream.
53         order Order
54         write func(*encoder, uint32) error
55         bits uint32
56         nBits uint
57         width uint
58         // litWidth is the width in bits of literal codes.
59         litWidth uint
60         // hi is the code implied by the next code emission.
61         // overflow is the code at which hi overflows the code
62         hi, overflow uint32
63         // savedCode is the accumulated code at the end of a
64         // call. It is equal to invalidCode if there was no
65         // savedCode uint32
66         // err is the first error encountered during writing
67         // will make any future Write calls return errClosed
68         err error
69         // table is the hash table from 20-bit keys to 12-bit
70         // entry contains key<<12|val and collisions resolve
71         // The keys consist of a 12-bit code prefix and an 8
72         // The values are a 12-bit code.
73         table [tableSize]uint32
74     }
75
76     // writeLSB writes the code c for "Least Significant Bits first"
77     func (e *encoder) writeLSB(c uint32) error {
78         e.bits |= c << e.nBits
79         e.nBits += e.width
80         for e.nBits >= 8 {
81             if err := e.w.WriteByte(uint8(e.bits)); err != nil {
82                 return err
83             }
84             e.bits >>= 8
85             e.nBits -= 8
86         }
87         return nil
88     }
89
90     // writeMSB writes the code c for "Most Significant Bits first"
91     func (e *encoder) writeMSB(c uint32) error {

```

```

92         e.bits |= c << (32 - e.width - e.nBits)
93         e.nBits += e.width
94         for e.nBits >= 8 {
95             if err := e.w.WriteByte(uint8(e.bits >> 24))
96                 return err
97             }
98             e.bits <<= 8
99             e.nBits -= 8
100         }
101         return nil
102     }
103
104     // errOutOfCodes is an internal error that means that the en
105     // of unused codes and a clear code needs to be sent next.
106     var errOutOfCodes = errors.New("lzw: out of codes")
107
108     // incHi increments e.hi and checks for both overflow and ru
109     // unused codes. In the latter case, incHi sends a clear cod
110     // encoder state and returns errOutOfCodes.
111     func (e *encoder) incHi() error {
112         e.hi++
113         if e.hi == e.overflow {
114             e.width++
115             e.overflow <<= 1
116         }
117         if e.hi == maxCode {
118             clear := uint32(1) << e.litWidth
119             if err := e.write(e, clear); err != nil {
120                 return err
121             }
122             e.width = uint(e.litWidth) + 1
123             e.hi = clear + 1
124             e.overflow = clear << 1
125             for i := range e.table {
126                 e.table[i] = invalidEntry
127             }
128             return errOutOfCodes
129         }
130         return nil
131     }
132
133     // Write writes a compressed representation of p to e's unde
134     func (e *encoder) Write(p []byte) (int, error) {
135         if e.err != nil {
136             return 0, e.err
137         }
138         if len(p) == 0 {
139             return 0, nil
140         }

```

```

141     litMask := uint32(1<<e.litWidth - 1)
142     code := e.savedCode
143     if code == invalidCode {
144         // The first code sent is always a literal c
145         code, p = uint32(p[0])&litMask, p[1:]
146     }
147 loop:
148     for _, x := range p {
149         literal := uint32(x) & litMask
150         key := code<<8 | literal
151         // If there is a hash table hit for this key
152         // and do not emit a code yet.
153         hash := (key>>12 ^ key) & tableMask
154         for h, t := hash, e.table[hash]; t != invali
155             if key == t>>12 {
156                 code = t & maxCode
157                 continue loop
158             }
159             h = (h + 1) & tableMask
160             t = e.table[h]
161         }
162         // Otherwise, write the current code, and li
163         // the next emitted code.
164         if e.err = e.write(e, code); e.err != nil {
165             return 0, e.err
166         }
167         code = literal
168         // Increment e.hi, the next implied code. If
169         // the encoder state (including clearing the
170         if err := e.incHi(); err != nil {
171             if err == errOutOfCodes {
172                 continue
173             }
174             e.err = err
175             return 0, e.err
176         }
177         // Otherwise, insert key -> e.hi into the ma
178         for {
179             if e.table[hash] == invalidEntry {
180                 e.table[hash] = (key << 12)
181                 break
182             }
183             hash = (hash + 1) & tableMask
184         }
185     }
186     e.savedCode = code
187     return len(p), nil
188 }
189

```

```

190 // Close closes the encoder, flushing any pending output. It
191 // flush e's underlying writer.
192 func (e *encoder) Close() error {
193     if e.err != nil {
194         if e.err == errClosed {
195             return nil
196         }
197         return e.err
198     }
199     // Make any future calls to Write return errClosed.
200     e.err = errClosed
201     // Write the savedCode if valid.
202     if e.savedCode != invalidCode {
203         if err := e.write(e, e.savedCode); err != nil {
204             return err
205         }
206         if err := e.incHi(); err != nil && err != errClosed {
207             return err
208         }
209     }
210     // Write the eof code.
211     eof := uint32(1) << e.litWidth + 1
212     if err := e.write(e, eof); err != nil {
213         return err
214     }
215     // Write the final bits.
216     if e.nBits > 0 {
217         if e.order == MSB {
218             e.bits >>= 24
219         }
220         if err := e.w.WriteByte(uint8(e.bits)); err != nil {
221             return err
222         }
223     }
224     return e.w.Flush()
225 }
226
227 // NewWriter creates a new io.WriteCloser that satisfies writing
228 // the data and writing it to w.
229 // It is the caller's responsibility to call Close on the Writer
230 // after finished writing.
231 // The number of bits to use for literal codes, litWidth, must be in
232 // range [2,8] and is typically 8.
233 func NewWriter(w io.Writer, order Order, litWidth int) io.WriteCloser {
234     var write func(*encoder, uint32) error
235     switch order {
236     case LSB:
237         write = (*encoder).writeLSB
238     case MSB:
239         write = (*encoder).writeMSB

```

```

240     default:
241         return &errWriteCloser{errors.New("lzw: unkn
242     }
243     if litWidth < 2 || 8 < litWidth {
244         return &errWriteCloser{fmt.Errorf("lzw: litw
245     }
246     bw, ok := w.(writer)
247     if !ok {
248         bw = bufio.NewWriter(w)
249     }
250     lw := uint(litWidth)
251     return &encoder{
252         w:         bw,
253         order:     order,
254         write:     write,
255         width:    1 + lw,
256         litWidth: lw,
257         hi:       1<<lw + 1,
258         overflow: 1 << (lw + 1),
259         savedCode: invalidCode,
260     }
261 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/zlib/reader.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6 Package zlib implements reading and writing of zlib format c
7 as specified in RFC 1950.
8
9 The implementation provides filters that uncompress during r
10 and compress during writing. For example, to write compress
11 to a buffer:
12
13     var b bytes.Buffer
14     w, err := zlib.NewWriter(&b)
15     w.Write([]byte("hello, world\n"))
16     w.Close()
17
18 and to read that data back:
19
20     r, err := zlib.NewReader(&b)
21     io.Copy(os.Stdout, r)
22     r.Close()
23 */
24 package zlib
25
26 import (
27     "bufio"
28     "compress/flate"
29     "errors"
30     "hash"
31     "hash/adler32"
32     "io"
33 )
34
35 const zlibDeflate = 8
36
37 var (
38     // ErrChecksum is returned when reading ZLIB data th
39     ErrChecksum = errors.New("zlib: invalid checksum")
40     // ErrDictionary is returned when reading ZLIB data
41     ErrDictionary = errors.New("zlib: invalid dictionary")
```

```

42         // ErrHeader is returned when reading ZLIB data that
43         ErrHeader = errors.New("zlib: invalid header")
44     )
45
46     type reader struct {
47         r          flate.Reader
48         decompressor io.ReadCloser
49         digest      hash.Hash32
50         err         error
51         scratch     [4]byte
52     }
53
54     // NewReader creates a new io.ReadCloser that satisfies read
55     // The implementation buffers input and may read more data t
56     // It is the caller's responsibility to call Close on the Re
57     func NewReader(r io.Reader) (io.ReadCloser, error) {
58         return NewReaderDict(r, nil)
59     }
60
61     // NewReaderDict is like NewReader but uses a preset diction
62     // NewReaderDict ignores the dictionary if the compressed da
63     func NewReaderDict(r io.Reader, dict []byte) (io.ReadCloser,
64         z := new(reader)
65         if fr, ok := r.(flate.Reader); ok {
66             z.r = fr
67         } else {
68             z.r = bufio.NewReader(r)
69         }
70         _, err := io.ReadFull(z.r, z.scratch[0:2])
71         if err != nil {
72             return nil, err
73         }
74         h := uint(z.scratch[0])<<8 | uint(z.scratch[1])
75         if (z.scratch[0]&0x0f != zlibDeflate) || (h%31 != 0)
76             return nil, ErrHeader
77         }
78         if z.scratch[1]&0x20 != 0 {
79             _, err = io.ReadFull(z.r, z.scratch[0:4])
80             if err != nil {
81                 return nil, err
82             }
83             checksum := uint32(z.scratch[0])<<24 | uint3
84             if checksum != adler32.Checksum(dict) {
85                 return nil, ErrDictionary
86             }
87             z.decompressor = flate.NewReaderDict(z.r, di
88         } else {
89             z.decompressor = flate.NewReader(z.r)
90         }
91         z.digest = adler32.New()

```

```

92         return z, nil
93     }
94
95     func (z *reader) Read(p []byte) (n int, err error) {
96         if z.err != nil {
97             return 0, z.err
98         }
99         if len(p) == 0 {
100            return 0, nil
101        }
102
103        n, err = z.decompressor.Read(p)
104        z.digest.Write(p[0:n])
105        if n != 0 || err != io.EOF {
106            z.err = err
107            return
108        }
109
110        // Finished file; check checksum.
111        if _, err := io.ReadFull(z.r, z.scratch[0:4]); err !=
112            z.err = err
113            return 0, err
114        }
115        // ZLIB (RFC 1950) is big-endian, unlike GZIP (RFC 1
116        checksum := uint32(z.scratch[0])<<24 | uint32(z.scr
117        if checksum != z.digest.Sum32() {
118            z.err = ErrChecksum
119            return 0, z.err
120        }
121        return
122    }
123
124    // Calling Close does not close the wrapped io.Reader origin
125    func (z *reader) Close() error {
126        if z.err != nil {
127            return z.err
128        }
129        z.err = z.decompressor.Close()
130        return z.err
131    }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/compress/zlib/writer.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package zlib
6
7 import (
8     "compress/flate"
9     "fmt"
10    "hash"
11    "hash/adler32"
12    "io"
13 )
14
15 // These constants are copied from the flate package, so tha
16 // "compress/zlib" does not also have to import "compress/fl
17 const (
18     NoCompression      = flate.NoCompression
19     BestSpeed           = flate.BestSpeed
20     BestCompression    = flate.BestCompression
21     DefaultCompression = flate.DefaultCompression
22 )
23
24 // A Writer takes data written to it and writes the compress
25 // form of that data to an underlying writer (see NewWriter)
26 type Writer struct {
27     w          io.Writer
28     level     int
29     dict      []byte
30     compressor *flate.Writer
31     digest    hash.Hash32
32     err       error
33     scratch   [4]byte
34     wroteHeader bool
35 }
36
37 // NewWriter creates a new Writer that satisfies writes by c
38 // written to w.
39 //
40 // It is the caller's responsibility to call Close on the Wr
41 // Writes may be buffered and not flushed until Close.
```

```

42 func NewWriter(w io.Writer) *Writer {
43     z, _ := NewWriterLevelDict(w, DefaultCompression, nil)
44     return z
45 }
46
47 // NewWriterLevel is like NewWriter but specifies the compression
48 // of assuming DefaultCompression.
49 //
50 // The compression level can be DefaultCompression, NoCompression,
51 // integer value between BestSpeed and BestCompression inclusive.
52 // returned will be nil if the level is valid.
53 func NewWriterLevel(w io.Writer, level int) (*Writer, error) {
54     return NewWriterLevelDict(w, level, nil)
55 }
56
57 // NewWriterLevelDict is like NewWriterLevel but specifies a dictionary
58 // to compress with.
59 //
60 // The dictionary may be nil. If not, its contents should not be
61 // the Writer is closed.
62 func NewWriterLevelDict(w io.Writer, level int, dict []byte) (*Writer, error) {
63     if level < DefaultCompression || level > BestCompression {
64         return nil, fmt.Errorf("zlib: invalid compression level")
65     }
66     return &Writer{
67         w:      w,
68         level:  level,
69         dict:   dict,
70     }, nil
71 }
72
73 // writeHeader writes the ZLIB header.
74 func (z *Writer) writeHeader() (err error) {
75     z.wroteHeader = true
76     // ZLIB has a two-byte header (as documented in RFC 1950)
77     // The first four bits is the CINFO (compression info)
78     // The next four bits is the CM (compression method)
79     z.scratch[0] = 0x78
80     // The next two bits is the FLEVEL (compression level)
81     // 0=fastest, 1=fast, 2=default, 3=best.
82     // The next bit, FDICT, is set if a dictionary is given
83     // The final five FCHECK bits form a mod-31 checksum
84     switch z.level {
85     case 0, 1:
86         z.scratch[1] = 0 << 6
87     case 2, 3, 4, 5:
88         z.scratch[1] = 1 << 6
89     case 6, -1:
90         z.scratch[1] = 2 << 6
91     case 7, 8, 9:

```

```

92             z.scratch[1] = 3 << 6
93 default:
94             panic("unreachable")
95     }
96     if z.dict != nil {
97         z.scratch[1] |= 1 << 5
98     }
99     z.scratch[1] += uint8(31 - (uint16(z.scratch[0])<<8+
100 if _, err = z.w.Write(z.scratch[0:2]); err != nil {
101     return err
102 }
103 if z.dict != nil {
104     // The next four bytes are the Adler-32 chec
105     checksum := adler32.Checksum(z.dict)
106     z.scratch[0] = uint8(checksum >> 24)
107     z.scratch[1] = uint8(checksum >> 16)
108     z.scratch[2] = uint8(checksum >> 8)
109     z.scratch[3] = uint8(checksum >> 0)
110     if _, err = z.w.Write(z.scratch[0:4]); err !
111         return err
112     }
113 }
114 z.compressor, err = flate.NewWriterDict(z.w, z.level
115 if err != nil {
116     return err
117 }
118 z.digest = adler32.New()
119 return nil
120 }
121
122 // Write writes a compressed form of p to the underlying io.
123 // compressed bytes are not necessarily flushed until the wr
124 // explicitly flushed.
125 func (z *Writer) Write(p []byte) (n int, err error) {
126     if !z.wroteHeader {
127         z.err = z.writeHeader()
128     }
129     if z.err != nil {
130         return 0, z.err
131     }
132     if len(p) == 0 {
133         return 0, nil
134     }
135     n, err = z.compressor.Write(p)
136     if err != nil {
137         z.err = err
138         return
139     }
140     z.digest.Write(p)

```

```

141         return
142     }
143
144     // Flush flushes the Writer to its underlying io.Writer.
145     func (z *Writer) Flush() error {
146         if !z.wroteHeader {
147             z.err = z.writeHeader()
148         }
149         if z.err != nil {
150             return z.err
151         }
152         z.err = z.compressor.Flush()
153         return z.err
154     }
155
156     // Calling Close does not close the wrapped io.Writer origin
157     func (z *Writer) Close() error {
158         if !z.wroteHeader {
159             z.err = z.writeHeader()
160         }
161         if z.err != nil {
162             return z.err
163         }
164         z.err = z.compressor.Close()
165         if z.err != nil {
166             return z.err
167         }
168         checksum := z.digest.Sum32()
169         // ZLIB (RFC 1950) is big-endian, unlike GZIP (RFC 1
170         z.scratch[0] = uint8(checksum >> 24)
171         z.scratch[1] = uint8(checksum >> 16)
172         z.scratch[2] = uint8(checksum >> 8)
173         z.scratch[3] = uint8(checksum >> 0)
174         _, z.err = z.w.Write(z.scratch[0:4])
175         return z.err
176     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/container/heap/heap.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package heap provides heap operations for any type that i
6 // heap.Interface. A heap is a tree with the property that e
7 // highest-valued node in its subtree.
8 //
9 // A heap is a common way to implement a priority queue. To
10 // queue, implement the Heap interface with the (negative) p
11 // ordering for the Less method, so Push adds items while Po
12 // highest-priority item from the queue. The Examples includ
13 // implementation; the file example_test.go has the complete
14 //
15 package heap
16
17 import "sort"
18
19 // Any type that implements heap.Interface may be used as a
20 // min-heap with the following invariants (established after
21 // Init has been called or if the data is empty or sorted):
22 //
23 //     !h.Less(j, i) for 0 <= i < h.Len() and j = 2*i+1 or
24 //
25 // Note that Push and Pop in this interface are for package
26 // implementation to call. To add and remove things from th
27 // use heap.Push and heap.Pop.
28 type Interface interface {
29     sort.Interface
30     Push(x interface{}) // add x as element Len()
31     Pop() interface{}   // remove and return element Len
32 }
33
34 // A heap must be initialized before any of the heap operati
35 // can be used. Init is idempotent with respect to the heap
36 // and may be called whenever the heap invariants may have b
37 // Its complexity is O(n) where n = h.Len().
38 //
39 func Init(h Interface) {
40     // heapify
41     n := h.Len()
```

```

42         for i := n/2 - 1; i >= 0; i-- {
43             down(h, i, n)
44         }
45     }
46
47     // Push pushes the element x onto the heap. The complexity is
48     // O(log(n)) where n = h.Len().
49     //
50     func Push(h Interface, x interface{}) {
51         h.Push(x)
52         up(h, h.Len()-1)
53     }
54
55     // Pop removes the minimum element (according to Less) from
56     // and returns it. The complexity is O(log(n)) where n = h.L
57     // Same as Remove(h, 0).
58     //
59     func Pop(h Interface) interface{} {
60         n := h.Len() - 1
61         h.Swap(0, n)
62         down(h, 0, n)
63         return h.Pop()
64     }
65
66     // Remove removes the element at index i from the heap.
67     // The complexity is O(log(n)) where n = h.Len().
68     //
69     func Remove(h Interface, i int) interface{} {
70         n := h.Len() - 1
71         if n != i {
72             h.Swap(i, n)
73             down(h, i, n)
74             up(h, i)
75         }
76         return h.Pop()
77     }
78
79     func up(h Interface, j int) {
80         for {
81             i := (j - 1) / 2 // parent
82             if i == j || h.Less(i, j) {
83                 break
84             }
85             h.Swap(i, j)
86             j = i
87         }
88     }
89
90     func down(h Interface, i, n int) {
91         for {

```

```
92         j1 := 2*i + 1
93         if j1 >= n {
94             break
95         }
96         j := j1 // left child
97         if j2 := j1 + 1; j2 < n && !h.Less(j1, j2) {
98             j = j2 // = 2*i + 2 // right child
99         }
100        if h.Less(i, j) {
101            break
102        }
103        h.Swap(i, j)
104        i = j
105    }
106 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/container/list/list.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package list implements a doubly linked list.
6 //
7 // To iterate over a list (where l is a *List):
8 //     for e := l.Front(); e != nil; e = e.Next() {
9 //         // do something with e.Value
10 //     }
11 //
12 package list
13
14 // Element is an element in the linked list.
15 type Element struct {
16     // Next and previous pointers in the doubly-linked list
17     // The front of the list has prev = nil, and the back
18     // has next = nil.
19     next, prev *Element
20
21     // The list to which this element belongs.
22     list *List
23
24     // The contents of this list element.
25     Value interface{}
26 }
27
28 // Next returns the next list element or nil.
29 func (e *Element) Next() *Element { return e.next }
30
31 // Prev returns the previous list element or nil.
32 func (e *Element) Prev() *Element { return e.prev }
33
34 // List represents a doubly linked list.
35 // The zero value for List is an empty list ready to use.
36 type List struct {
37     front, back *Element
38     len        int
39 }
40
41 // Init initializes or clears a List.
```

```

42         l.front = nil
43         l.back = nil
44         l.len = 0
45         return l
46     }
47
48     // New returns an initialized list.
49     func New() *List { return new(List) }
50
51     // Front returns the first element in the list.
52     func (l *List) Front() *Element { return l.front }
53
54     // Back returns the last element in the list.
55     func (l *List) Back() *Element { return l.back }
56
57     // Remove removes the element from the list
58     // and returns its Value.
59     func (l *List) Remove(e *Element) interface{} {
60         l.remove(e)
61         e.list = nil // do what remove does not
62         return e.Value
63     }
64
65     // remove the element from the list, but do not clear the El
66     // This is so that other List methods may use remove when re
67     // without needing to restore the list field.
68     func (l *List) remove(e *Element) {
69         if e.list != l {
70             return
71         }
72         if e.prev == nil {
73             l.front = e.next
74         } else {
75             e.prev.next = e.next
76         }
77         if e.next == nil {
78             l.back = e.prev
79         } else {
80             e.next.prev = e.prev
81         }
82
83         e.prev = nil
84         e.next = nil
85         l.len--
86     }
87
88     func (l *List) insertBefore(e *Element, mark *Element) {
89         if mark.prev == nil {
90             // new front of the list
91             l.front = e

```

```

92         } else {
93             mark.prev.next = e
94         }
95         e.prev = mark.prev
96         mark.prev = e
97         e.next = mark
98         l.len++
99     }
100
101     func (l *List) insertAfter(e *Element, mark *Element) {
102         if mark.next == nil {
103             // new back of the list
104             l.back = e
105         } else {
106             mark.next.prev = e
107         }
108         e.next = mark.next
109         mark.next = e
110         e.prev = mark
111         l.len++
112     }
113
114     func (l *List) insertFront(e *Element) {
115         if l.front == nil {
116             // empty list
117             l.front, l.back = e, e
118             e.prev, e.next = nil, nil
119             l.len = 1
120             return
121         }
122         l.insertBefore(e, l.front)
123     }
124
125     func (l *List) insertBack(e *Element) {
126         if l.back == nil {
127             // empty list
128             l.front, l.back = e, e
129             e.prev, e.next = nil, nil
130             l.len = 1
131             return
132         }
133         l.insertAfter(e, l.back)
134     }
135
136     // PushFront inserts the value at the front of the list and
137     func (l *List) PushFront(value interface{}) *Element {
138         e := &Element{nil, nil, l, value}
139         l.insertFront(e)
140         return e

```

```

141 }
142
143 // PushBack inserts the value at the back of the list and re
144 func (l *List) PushBack(value interface{}) *Element {
145     e := &Element{nil, nil, l, value}
146     l.insertBack(e)
147     return e
148 }
149
150 // InsertBefore inserts the value immediately before mark an
151 func (l *List) InsertBefore(value interface{}, mark *Element)
152     if mark.list != l {
153         return nil
154     }
155     e := &Element{nil, nil, l, value}
156     l.insertBefore(e, mark)
157     return e
158 }
159
160 // InsertAfter inserts the value immediately after mark and
161 func (l *List) InsertAfter(value interface{}, mark *Element)
162     if mark.list != l {
163         return nil
164     }
165     e := &Element{nil, nil, l, value}
166     l.insertAfter(e, mark)
167     return e
168 }
169
170 // MoveToFront moves the element to the front of the list.
171 func (l *List) MoveToFront(e *Element) {
172     if e.list != l || l.front == e {
173         return
174     }
175     l.remove(e)
176     l.insertFront(e)
177 }
178
179 // MoveToBack moves the element to the back of the list.
180 func (l *List) MoveToBack(e *Element) {
181     if e.list != l || l.back == e {
182         return
183     }
184     l.remove(e)
185     l.insertBack(e)
186 }
187
188 // Len returns the number of elements in the list.
189 func (l *List) Len() int { return l.len }

```

```

190
191 // PushBackList inserts each element of ol at the back of th
192 func (l *List) PushBackList(ol *List) {
193     last := ol.Back()
194     for e := ol.Front(); e != nil; e = e.Next() {
195         l.PushBack(e.Value)
196         if e == last {
197             break
198         }
199     }
200 }
201
202 // PushFrontList inserts each element of ol at the front of
203 func (l *List) PushFrontList(ol *List) {
204     first := ol.Front()
205     for e := ol.Back(); e != nil; e = e.Prev() {
206         l.PushFront(e.Value)
207         if e == first {
208             break
209         }
210     }
211 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/container/ring/ring.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package ring implements operations on circular lists.
6 package ring
7
8 // A Ring is an element of a circular list, or ring.
9 // Rings do not have a beginning or end; a pointer to any ri
10 // serves as reference to the entire ring. Empty rings are r
11 // as nil Ring pointers. The zero value for a Ring is a one-
12 // ring with a nil Value.
13 //
14 type Ring struct {
15     next, prev *Ring
16     Value      interface{} // for use by client; untouched
17 }
18
19 func (r *Ring) init() *Ring {
20     r.next = r
21     r.prev = r
22     return r
23 }
24
25 // Next returns the next ring element. r must not be empty.
26 func (r *Ring) Next() *Ring {
27     if r.next == nil {
28         return r.init()
29     }
30     return r.next
31 }
32
33 // Prev returns the previous ring element. r must not be emp
34 func (r *Ring) Prev() *Ring {
35     if r.next == nil {
36         return r.init()
37     }
38     return r.prev
39 }
40
41 // Move moves n % r.Len() elements backward (n < 0) or forwa
```

```

42 // in the ring and returns that ring element. r must not be
43 //
44 func (r *Ring) Move(n int) *Ring {
45     if r.next == nil {
46         return r.init()
47     }
48     switch {
49     case n < 0:
50         for ; n < 0; n++ {
51             r = r.prev
52         }
53     case n > 0:
54         for ; n > 0; n-- {
55             r = r.next
56         }
57     }
58     return r
59 }
60
61 // New creates a ring of n elements.
62 func New(n int) *Ring {
63     if n <= 0 {
64         return nil
65     }
66     r := new(Ring)
67     p := r
68     for i := 1; i < n; i++ {
69         p.next = &Ring{prev: p}
70         p = p.next
71     }
72     p.next = r
73     r.prev = p
74     return r
75 }
76
77 // Link connects ring r with with ring s such that r.Next()
78 // becomes s and returns the original value for r.Next().
79 // r must not be empty.
80 //
81 // If r and s point to the same ring, linking
82 // them removes the elements between r and s from the ring.
83 // The removed elements form a subring and the result is a
84 // reference to that subring (if no elements were removed,
85 // the result is still the original value for r.Next(),
86 // and not nil).
87 //
88 // If r and s point to different rings, linking
89 // them creates a single ring with the elements of s inserte
90 // after r. The result points to the element following the
91 // last element of s after insertion.

```

```

92 //
93 func (r *Ring) Link(s *Ring) *Ring {
94     n := r.Next()
95     if s != nil {
96         p := s.Prev()
97         // Note: Cannot use multiple assignment beca
98         // evaluation order of LHS is not specified.
99         r.next = s
100        s.prev = r
101        n.prev = p
102        p.next = n
103    }
104    return n
105 }
106
107 // Unlink removes n % r.Len() elements from the ring r, star
108 // at r.Next(). If n % r.Len() == 0, r remains unchanged.
109 // The result is the removed subring. r must not be empty.
110 //
111 func (r *Ring) Unlink(n int) *Ring {
112     if n <= 0 {
113         return nil
114     }
115     return r.Link(r.Move(n + 1))
116 }
117
118 // Len computes the number of elements in ring r.
119 // It executes in time proportional to the number of element
120 //
121 func (r *Ring) Len() int {
122     n := 0
123     if r != nil {
124         n = 1
125         for p := r.Next(); p != r; p = p.next {
126             n++
127         }
128     }
129     return n
130 }
131
132 // Do calls function f on each element of the ring, in forwa
133 // The behavior of Do is undefined if f changes *r.
134 func (r *Ring) Do(f func(interface{})) {
135     if r != nil {
136         f(r.Value)
137         for p := r.Next(); p != r; p = p.next {
138             f(p.Value)
139         }
140     }

```

141 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/crypto/crypto.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package crypto collects common cryptographic constants.
6 package crypto
7
8 import (
9     "hash"
10 )
11
12 // Hash identifies a cryptographic hash function that is imp
13 // package.
14 type Hash uint
15
16 const (
17     MD4          Hash = 1 + iota // import code.google.com/
18     MD5          // import crypto/md5
19     SHA1         // import crypto/sha1
20     SHA224       // import crypto/sha256
21     SHA256       // import crypto/sha256
22     SHA384       // import crypto/sha512
23     SHA512       // import crypto/sha512
24     MD5SHA1      // no implementation; MD5+
25     RIPEMD160    // import code.google.com/
26     maxHash
27 )
28
29 var digestSizes = []uint8{
30     MD4:      16,
31     MD5:      16,
32     SHA1:     20,
33     SHA224:   28,
34     SHA256:   32,
35     SHA384:   48,
36     SHA512:   64,
37     MD5SHA1:  36,
38     RIPEMD160: 20,
39 }
40
41 // Size returns the length, in bytes, of a digest resulting
42 // function. It doesn't require that the hash function in qu
43 // into the program.
44 func (h Hash) Size() int {
```

```

45         if h > 0 && h < maxHash {
46             return int(digestSizes[h])
47         }
48         panic("crypto: Size of unknown hash function")
49     }
50
51     var hashes = make([]func() hash.Hash, maxHash)
52
53     // New returns a new hash.Hash calculating the given hash fu
54     // if the hash function is not linked into the binary.
55     func (h Hash) New() hash.Hash {
56         if h > 0 && h < maxHash {
57             f := hashes[h]
58             if f != nil {
59                 return f()
60             }
61         }
62         panic("crypto: requested hash function is unavailabl
63     }
64
65     // Available reports whether the given hash function is link
66     func (h Hash) Available() bool {
67         return h < maxHash && hashes[h] != nil
68     }
69
70     // RegisterHash registers a function that returns a new inst
71     // hash function. This is intended to be called from the ini
72     // packages that implement hash functions.
73     func RegisterHash(h Hash, f func() hash.Hash) {
74         if h >= maxHash {
75             panic("crypto: RegisterHash of unknown hash
76         }
77         hashes[h] = f
78     }
79
80     // PrivateKey represents a private key using an unspecified
81     type PrivateKey interface{}

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/aes/block.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This Go implementation is derived in part from the refere
6 // ANSI C implementation, which carries the following notice
7 //
8 //     rijndael-alg-fst.c
9 //
10 //     @version 3.0 (December 2000)
11 //
12 //     Optimised ANSI C code for the Rijndael cipher (now A
13 //
14 //     @author Vincent Rijmen <vincent.rijmen@esat.kuleuven
15 //     @author Antoon Bosselaers <antoon.bosselaers@esat.ku
16 //     @author Paulo Barreto <paulo.barreto@terra.com.br>
17 //
18 //     This code is hereby placed in the public domain.
19 //
20 //     THIS SOFTWARE IS PROVIDED BY THE AUTHORS 'AS IS' A
21 //     OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO
22 //     WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PART
23 //     ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CO
24 //     LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL
25 //     CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO
26 //     SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
27 //     BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THE
28 //     WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCL
29 //     OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF T
30 //     EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
31 //
32 // See FIPS 197 for specification, and see Daemen and Rijmen
33 // for implementation details.
34 //     http://www.csrc.nist.gov/publications/fips/fips197/f
35 //     http://csrc.nist.gov/archive/aes/rijndael/Rijndael-a
36
37 package aes
38
39 // Encrypt one block from src into dst, using the expanded k
40 func encryptBlock(xk []uint32, dst, src []byte) {
41     var s0, s1, s2, s3, t0, t1, t2, t3 uint32
```

```

42
43     s0 = uint32(src[0])<<24 | uint32(src[1])<<16 | uint3
44     s1 = uint32(src[4])<<24 | uint32(src[5])<<16 | uint3
45     s2 = uint32(src[8])<<24 | uint32(src[9])<<16 | uint3
46     s3 = uint32(src[12])<<24 | uint32(src[13])<<16 | uin
47
48     // First round just XORs input with key.
49     s0 ^= xk[0]
50     s1 ^= xk[1]
51     s2 ^= xk[2]
52     s3 ^= xk[3]
53
54     // Middle rounds shuffle using tables.
55     // Number of rounds is set by length of expanded key
56     nr := len(xk)/4 - 2 // - 2: one above, one more belo
57     k := 4
58     for r := 0; r < nr; r++ {
59         t0 = xk[k+0] ^ te0[uint8(s0>>24)] ^ te1[uint
60         t1 = xk[k+1] ^ te0[uint8(s1>>24)] ^ te1[uint
61         t2 = xk[k+2] ^ te0[uint8(s2>>24)] ^ te1[uint
62         t3 = xk[k+3] ^ te0[uint8(s3>>24)] ^ te1[uint
63         k += 4
64         s0, s1, s2, s3 = t0, t1, t2, t3
65     }
66
67     // Last round uses s-box directly and XORs to produc
68     s0 = uint32(sbox0[t0>>24])<<24 | uint32(sbox0[t1>>16
69     s1 = uint32(sbox0[t1>>24])<<24 | uint32(sbox0[t2>>16
70     s2 = uint32(sbox0[t2>>24])<<24 | uint32(sbox0[t3>>16
71     s3 = uint32(sbox0[t3>>24])<<24 | uint32(sbox0[t0>>16
72
73     s0 ^= xk[k+0]
74     s1 ^= xk[k+1]
75     s2 ^= xk[k+2]
76     s3 ^= xk[k+3]
77
78     dst[0], dst[1], dst[2], dst[3] = byte(s0>>24), byte(
79     dst[4], dst[5], dst[6], dst[7] = byte(s1>>24), byte(
80     dst[8], dst[9], dst[10], dst[11] = byte(s2>>24), byt
81     dst[12], dst[13], dst[14], dst[15] = byte(s3>>24), b
82 }
83
84 // Decrypt one block from src into dst, using the expanded k
85 func decryptBlock(xk []uint32, dst, src []byte) {
86     var s0, s1, s2, s3, t0, t1, t2, t3 uint32
87
88     s0 = uint32(src[0])<<24 | uint32(src[1])<<16 | uint3
89     s1 = uint32(src[4])<<24 | uint32(src[5])<<16 | uint3
90     s2 = uint32(src[8])<<24 | uint32(src[9])<<16 | uint3
91     s3 = uint32(src[12])<<24 | uint32(src[13])<<16 | uin

```

```

92
93 // First round just XORs input with key.
94 s0 ^= xk[0]
95 s1 ^= xk[1]
96 s2 ^= xk[2]
97 s3 ^= xk[3]
98
99 // Middle rounds shuffle using tables.
100 // Number of rounds is set by length of expanded key
101 nr := len(xk)/4 - 2 // - 2: one above, one more below
102 k := 4
103 for r := 0; r < nr; r++ {
104     t0 = xk[k+0] ^ td0[uint8(s0>>24)] ^ td1[uint
105     t1 = xk[k+1] ^ td0[uint8(s1>>24)] ^ td1[uint
106     t2 = xk[k+2] ^ td0[uint8(s2>>24)] ^ td1[uint
107     t3 = xk[k+3] ^ td0[uint8(s3>>24)] ^ td1[uint
108     k += 4
109     s0, s1, s2, s3 = t0, t1, t2, t3
110 }
111
112 // Last round uses s-box directly and XORs to produce
113 s0 = uint32(sbox1[t0>>24])<<24 | uint32(sbox1[t3>>16]
114 s1 = uint32(sbox1[t1>>24])<<24 | uint32(sbox1[t0>>16]
115 s2 = uint32(sbox1[t2>>24])<<24 | uint32(sbox1[t1>>16]
116 s3 = uint32(sbox1[t3>>24])<<24 | uint32(sbox1[t2>>16]
117
118 s0 ^= xk[k+0]
119 s1 ^= xk[k+1]
120 s2 ^= xk[k+2]
121 s3 ^= xk[k+3]
122
123 dst[0], dst[1], dst[2], dst[3] = byte(s0>>24), byte(
124 dst[4], dst[5], dst[6], dst[7] = byte(s1>>24), byte(
125 dst[8], dst[9], dst[10], dst[11] = byte(s2>>24), byte(
126 dst[12], dst[13], dst[14], dst[15] = byte(s3>>24), b
127 }
128
129 // Apply sbox0 to each byte in w.
130 func subw(w uint32) uint32 {
131     return uint32(sbox0[w>>24])<<24 |
132         uint32(sbox0[w>>16&0xff])<<16 |
133         uint32(sbox0[w>>8&0xff])<<8 |
134         uint32(sbox0[w&0xff])
135 }
136
137 // Rotate
138 func rotw(w uint32) uint32 { return w<<8 | w>>24 }
139
140 // Key expansion algorithm. See FIPS-197, Figure 11.

```

```

141 // Their rcon[i] is our powx[i-1] << 24.
142 func expandKey(key []byte, enc, dec []uint32) {
143     // Encryption key setup.
144     var i int
145     nk := len(key) / 4
146     for i = 0; i < nk; i++ {
147         enc[i] = uint32(key[4*i])<<24 | uint32(key[4
148     }
149     for ; i < len(enc); i++ {
150         t := enc[i-1]
151         if i%nk == 0 {
152             t = subw(rotw(t)) ^ (uint32(powx[i/n
153         } else if nk > 6 && i%nk == 4 {
154             t = subw(t)
155         }
156         enc[i] = enc[i-nk] ^ t
157     }
158
159     // Derive decryption key from encryption key.
160     // Reverse the 4-word round key sets from enc to pro
161     // All sets but the first and last get the MixColumn
162     if dec == nil {
163         return
164     }
165     n := len(enc)
166     for i := 0; i < n; i += 4 {
167         ei := n - i - 4
168         for j := 0; j < 4; j++ {
169             x := enc[ei+j]
170             if i > 0 && i+4 < n {
171                 x = td0[sbox0[x>>24]] ^ td1[
172             }
173             dec[i+j] = x
174         }
175     }
176 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/aes/cipher.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package aes
6
7 import (
8     "crypto/cipher"
9     "strconv"
10 )
11
12 // The AES block size in bytes.
13 const BlockSize = 16
14
15 // A cipher is an instance of AES encryption using a particu
16 type aesCipher struct {
17     enc []uint32
18     dec []uint32
19 }
20
21 type KeySizeError int
22
23 func (k KeySizeError) Error() string {
24     return "crypto/aes: invalid key size " + strconv.Ito
25 }
26
27 // NewCipher creates and returns a new cipher.Block.
28 // The key argument should be the AES key,
29 // either 16, 24, or 32 bytes to select
30 // AES-128, AES-192, or AES-256.
31 func NewCipher(key []byte) (cipher.Block, error) {
32     k := len(key)
33     switch k {
34     default:
35         return nil, KeySizeError(k)
36     case 16, 24, 32:
37         break
38     }
39
40     n := k + 28
41     c := &aesCipher{make([]uint32, n), make([]uint32, n)}
```

```
42         expandKey(key, c.enc, c.dec)
43         return c, nil
44     }
45
46     func (c *aesCipher) BlockSize() int { return BlockSize }
47
48     func (c *aesCipher) Encrypt(dst, src []byte) { encryptBlock(
49
50     func (c *aesCipher) Decrypt(dst, src []byte) { decryptBlock(
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/aes/const.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package aes implements AES encryption (formerly Rijndael)
6 // U.S. Federal Information Processing Standards Publication
7 package aes
8
9 // This file contains AES constants - 8720 bytes of initiali
10
11 // http://www.csrc.nist.gov/publications/fips/fips197/fips-1
12
13 // AES is based on the mathematical behavior of binary polyn
14 // (polynomials over GF(2)) modulo the irreducible polynomia
15 // Addition of these binary polynomials corresponds to binar
16 // Reducing mod poly corresponds to binary xor with poly eve
17 // time a 0x100 bit appears.
18 const poly = 1<<8 | 1<<4 | 1<<3 | 1<<1 | 1<<0 // x8 + x4 + x
19
20 // Powers of x mod poly in GF(2).
21 var powx = [16]byte{
22     0x01,
23     0x02,
24     0x04,
25     0x08,
26     0x10,
27     0x20,
28     0x40,
29     0x80,
30     0x1b,
31     0x36,
32     0x6c,
33     0xd8,
34     0xab,
35     0x4d,
36     0x9a,
37     0x2f,
38 }
39
40 // FIPS-197 Figure 7. S-box substitution values in hexadecir
41 var sbox0 = [256]byte{
```

```

42         0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30
43         0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad
44         0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34
45         0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07
46         0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52
47         0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a
48         0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45
49         0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc
50         0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4
51         0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46
52         0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2
53         0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c
54         0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8
55         0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61
56         0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b
57         0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41
58     }
59
60     // FIPS-197 Figure 14. Inverse S-box substitution values in
61     var sbox1 = [256]byte{
62         0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf
63         0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34
64         0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee
65         0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76
66         0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4
67         0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e
68         0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7
69         0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1
70         0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97
71         0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2
72         0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f
73         0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a
74         0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1
75         0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d
76         0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8
77         0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1
78     }
79
80     // Lookup tables for encryption.
81     // These can be recomputed by adapting the tests in aes_test
82
83     var te0 = [256]uint32{
84         0xc66363a5, 0xf87c7c84, 0xee777799, 0xf67b7b8d, 0xff
85         0x60303050, 0x02010103, 0xce6767a9, 0x562b2b7d, 0xe7
86         0x8fcaca45, 0x1f82829d, 0x89c9c940, 0xfa7d7d87, 0xef
87         0x41adadec, 0xb3d4d467, 0x5fa2a2fd, 0x45afafea, 0x23
88         0x75b7b7c2, 0xefdfd1c, 0x3d9393ae, 0x4c26266a, 0x6c
89         0x6834345c, 0x51a5a5f4, 0xd1e5e534, 0xf9f1f108, 0xe2
90         0x0804040c, 0x95c7c752, 0x46232365, 0x9dc3c35e, 0x30
91         0x0e070709, 0x24121236, 0x1b80809b, 0xdfe2e23d, 0xcd

```

```

92      0x1209091b, 0x1d83839e, 0x582c2c74, 0x341a1a2e, 0x36
93      0xa45252f6, 0x763b3b4d, 0xb7d6d661, 0x7db3b3ce, 0x52
94      0xa65353f5, 0xb9d1d168, 0x00000000, 0xc1eded2c, 0x40
95      0xd46a6abe, 0x8dcbcb46, 0x67bebed9, 0x7239394b, 0x94
96      0xbbd0d06b, 0xc5efef2a, 0x4faaaae5, 0xedfbfb16, 0x86
97      0x8a4545cf, 0xe9f9f910, 0x04020206, 0xfe7f7f81, 0xa0
98      0xa25151f3, 0x5da3a3fe, 0x804040c0, 0x058f8f8a, 0x3f
99      0x63bcbcdf, 0x77b6b6c1, 0xafdada75, 0x42212163, 0x20
100     0x81cdcd4c, 0x180c0c14, 0x26131335, 0xc3ecec2f, 0xbe
101     0x93c4c457, 0x55a7a7f2, 0xfc7e7e82, 0x7a3d3d47, 0xc8
102     0xc06060a0, 0x19818198, 0x9e4f4fd1, 0xa3dcdc7f, 0x44
103     0x8c4646ca, 0xc7eeee29, 0x6bb8b8d3, 0x2814143c, 0xa7
104     0xdb0e0e3b, 0x64323256, 0x743a3a4e, 0x140a0a1e, 0x92
105     0x9fc2c25d, 0xbdd3d36e, 0x43acacef, 0xc46262a6, 0x39
106     0xd5e7e732, 0x8bc8c843, 0x6e373759, 0xda6d6db7, 0x01
107     0xd86c6cb4, 0xac5656fa, 0xf3f4f407, 0xcfeaea25, 0xca
108     0x6fbabad5, 0xf0787888, 0x4a25256f, 0x5c2e2e72, 0x38
109     0xcbe8e823, 0xa1dddd7c, 0xe874749c, 0x3e1f1f21, 0x96
110     0xe0707090, 0x7c3e3e42, 0x71b5b5c4, 0xcc6666aa, 0x90
111     0xc26161a3, 0x6a35355f, 0xae5757f9, 0x69b9b9d0, 0x17
112     0xd9e1e138, 0xebf8f813, 0x2b9898b3, 0x22111133, 0xd2
113     0x2d9b9bb6, 0x3c1e1e22, 0x15878792, 0xc9e9e920, 0x87
114     0x038c8c8f, 0x59a1a1f8, 0x09898980, 0x1a0d0d17, 0x65
115     0x824141c3, 0x299999b0, 0x5a2d2d77, 0x1e0f0f11, 0x7b
116 }
117 var te1 = [256]uint32{
118     0xa5c66363, 0x84f87c7c, 0x99ee7777, 0x8df67b7b, 0x0d
119     0x50603030, 0x03020101, 0xa9ce6767, 0x7d562b2b, 0x19
120     0x458fcaca, 0x9d1f8282, 0x4089c9c9, 0x87fa7d7d, 0x15
121     0xec41adad, 0x67b3d4d4, 0xfd5fa2a2, 0xea45afaf, 0xbf
122     0xc275b7b7, 0x1ce1fdfd, 0xae3d9393, 0x6a4c2626, 0x5a
123     0x5c683434, 0xf451a5a5, 0x34d1e5e5, 0x08f9f1f1, 0x93
124     0x0c080404, 0x5295c7c7, 0x65462323, 0x5e9dc3c3, 0x28
125     0x090e0707, 0x36241212, 0x9b1b8080, 0x3ddfe2e2, 0x26
126     0x1b120909, 0x9e1d8383, 0x74582c2c, 0x2e341a1a, 0x2d
127     0xf6a45252, 0x4d763b3b, 0x61b7d6d6, 0xce7db3b3, 0x7b
128     0xf5a65353, 0x68b9d1d1, 0x00000000, 0x2cc1eded, 0x60
129     0xbed46a6a, 0x468dcbcb, 0xd967bebe, 0x4b723939, 0xde
130     0x6bbbd0d0, 0x2ac5efef, 0xe54faaaa, 0x16edfbfb, 0xc5
131     0xcf8a4545, 0x10e9f9f9, 0x06040202, 0x81fe7f7f, 0xf0
132     0xf3a25151, 0xfe5da3a3, 0xc0804040, 0x8a058f8f, 0xad
133     0xdf63bcbcb, 0xc177b6b6, 0x75afdada, 0x63422121, 0x30
134     0x4c81cdcd, 0x14180c0c, 0x35261313, 0x2fc3ecec, 0xe1
135     0x5793c4c4, 0xf255a7a7, 0x82fc7e7e, 0x477a3d3d, 0xac
136     0xa0c06060, 0x98198181, 0xd19e4f4f, 0x7fa3dcdc, 0x66
137     0xca8c4646, 0x29c7eeee, 0xd36bb8b8, 0x3c281414, 0x79
138     0x3dbde0e0, 0x56643232, 0x4e743a3a, 0x1e140a0a, 0xdb
139     0x5d9fc2c2, 0x6ebdd3d3, 0xef43acac, 0xa6c46262, 0xa8
140     0x32d5e7e7, 0x438bc8c8, 0x596e3737, 0xb7da6d6d, 0x8c

```

```

141         0xb4d86c6c, 0xfaac5656, 0x07f3f4f4, 0x25cfeaea, 0xaf
142         0xd56fbaba, 0x88f07878, 0x6f4a2525, 0x725c2e2e, 0x24
143         0x23cbe8e8, 0x7ca1dddd, 0x9ce87474, 0x213e1f1f, 0xdd
144         0x90e07070, 0x427c3e3e, 0xc471b5b5, 0xaaacc666, 0xd8
145         0xa3c26161, 0x5f6a3535, 0xf9ae5757, 0xd069b9b9, 0x91
146         0x38d9e1e1, 0x13ebf8f8, 0xb32b9898, 0x33221111, 0xbb
147         0xb62d9b9b, 0x223c1e1e, 0x92158787, 0x20c9e9e9, 0x49
148         0x8f038c8c, 0xf859a1a1, 0x80098989, 0x171a0d0d, 0xda
149         0xc3824141, 0xb0299999, 0x775a2d2d, 0x111e0f0f, 0xcb
150     }
151     var te2 = [256]uint32{
152         0x63a5c663, 0x7c84f87c, 0x7799ee77, 0x7b8df67b, 0xf2
153         0x30506030, 0x01030201, 0x67a9ce67, 0x2b7d562b, 0xfe
154         0xca458fca, 0x829d1f82, 0xc94089c9, 0x7d87fa7d, 0xfa
155         0xadec41ad, 0xd467b3d4, 0xa2fd5fa2, 0xafea45af, 0x9c
156         0xb7c275b7, 0xfd1ce1fd, 0x93ae3d93, 0x266a4c26, 0x36
157         0x345c6834, 0xa5f451a5, 0xe534d1e5, 0xf108f9f1, 0x71
158         0x040c0804, 0xc75295c7, 0x23654623, 0xc35e9dc3, 0x18
159         0x07090e07, 0x12362412, 0x809b1b80, 0xe23ddfe2, 0xeb
160         0x091b1209, 0x839e1d83, 0x2c74582c, 0x1a2e341a, 0x1b
161         0x52f6a452, 0x3b4d763b, 0xd661b7d6, 0xb3ce7db3, 0x29
162         0x53f5a653, 0xd168b9d1, 0x00000000, 0xed2cc1ed, 0x20
163         0x6abed46a, 0xcb468dcb, 0xbed967be, 0x394b7239, 0x4a
164         0xd06bbbd0, 0xef2ac5ef, 0xaae54faa, 0xfb16edfb, 0x43
165         0x45cf8a45, 0xf910e9f9, 0x02060402, 0x7f81fe7f, 0x50
166         0x51f3a251, 0xa3fe5da3, 0x40c08040, 0x8f8a058f, 0x92
167         0xbcdf63bc, 0xb6c177b6, 0xda75afda, 0x21634221, 0x10
168         0xcd4c81cd, 0x0c14180c, 0x13352613, 0xec2fc3ec, 0x5f
169         0xc45793c4, 0xa7f255a7, 0x7e82fc7e, 0x3d477a3d, 0x64
170         0x60a0c060, 0x81981981, 0x4fd19e4f, 0xdc7fa3dc, 0x22
171         0x46ca8c46, 0xee29c7ee, 0xb8d36bb8, 0x143c2814, 0xde
172         0xe03bdbe0, 0x32566432, 0x3a4e743a, 0x0a1e140a, 0x49
173         0xc25d9fc2, 0xd36ebdd3, 0xacef43ac, 0x62a6c462, 0x91
174         0xe732d5e7, 0xc8438bc8, 0x37596e37, 0x6db7da6d, 0x8d
175         0x6cb4d86c, 0x56faac56, 0xf407f3f4, 0xea25cfea, 0x65
176         0xbad56fba, 0x7888f078, 0x256f4a25, 0x2e725c2e, 0x1c
177         0xe823cbe8, 0xdd7ca1dd, 0x749ce874, 0x1f213e1f, 0x4b
178         0x7090e070, 0x3e427c3e, 0xb5c471b5, 0x66aacc66, 0x48
179         0x61a3c261, 0x355f6a35, 0x57f9ae57, 0xb9d069b9, 0x86
180         0xe138d9e1, 0xf813ebf8, 0x98b32b98, 0x11332211, 0x69
181         0x9bb62d9b, 0x1e223c1e, 0x87921587, 0xe920c9e9, 0xce
182         0x8c8f038c, 0xa1f859a1, 0x89800989, 0x0d171a0d, 0xbf
183         0x41c38241, 0x99b02999, 0x2d775a2d, 0xf111e0f, 0xb0
184     }
185     var te3 = [256]uint32{
186         0x6363a5c6, 0x7c7c84f8, 0x777799ee, 0x7b7b8df6, 0xf2
187         0x30305060, 0x01010302, 0x6767a9ce, 0x2b2b7d56, 0xfe
188         0xcaca458f, 0x82829d1f, 0xc9c94089, 0x7d7d87fa, 0xfa
189         0xadadec41, 0xd4d467b3, 0xa2a2fd5f, 0xafafea45, 0x9c

```

```

190         0xb7b7c275, 0xfdfd1ce1, 0x9393ae3d, 0x26266a4c, 0x36
191         0x34345c68, 0xa5a5f451, 0xe5e534d1, 0xf1f108f9, 0x71
192         0x04040c08, 0xc7c75295, 0x23236546, 0xc3c35e9d, 0x18
193         0x0707090e, 0x12123624, 0x80809b1b, 0xe2e23ddf, 0xeb
194         0x09091b12, 0x83839e1d, 0x2c2c7458, 0x1a1a2e34, 0x1b
195         0x5252f6a4, 0x3b3b4d76, 0xd6d661b7, 0xb3b3ce7d, 0x29
196         0x5353f5a6, 0xd1d168b9, 0x00000000, 0xeded2cc1, 0x20
197         0x6a6abed4, 0xcbcb468d, 0xbebed967, 0x39394b72, 0x4a
198         0xd0d06bbb, 0xefef2ac5, 0xaaaae54f, 0xfbfb16ed, 0x43
199         0x4545cf8a, 0xf9f910e9, 0x02020604, 0x7f7f81fe, 0x50
200         0x5151f3a2, 0xa3a3fe5d, 0x4040c080, 0x8f8f8a05, 0x92
201         0xbcbcdf63, 0xb6b6c177, 0xdada75af, 0x21216342, 0x10
202         0xcdcd4c81, 0x0c0c1418, 0x13133526, 0xecec2fc3, 0x5f
203         0xc4c45793, 0xa7a7f255, 0x7e7e82fc, 0x3d3d477a, 0x64
204         0x6060a0c0, 0x81819819, 0x4f4fd19e, 0xdcdc7fa3, 0x22
205         0x4646ca8c, 0xeeee29c7, 0xb8b8d36b, 0x14143c28, 0xde
206         0xe0e03bdb, 0x32325664, 0x3a3a4e74, 0x0a0a1e14, 0x49
207         0xc2c25d9f, 0xd3d36ebd, 0xacacef43, 0x6262a6c4, 0x91
208         0xe7e732d5, 0xc8c8438b, 0x3737596e, 0x6d6db7da, 0x8d
209         0x6c6cb4d8, 0x5656faac, 0xf4f407f3, 0xeaea25cf, 0x65
210         0xbabad56f, 0x787888f0, 0x25256f4a, 0x2e2e725c, 0x1c
211         0xe8e823cb, 0xdddd7ca1, 0x74749ce8, 0x1f1f213e, 0x4b
212         0x707090e0, 0x3e3e427c, 0xb5b5c471, 0x6666aacc, 0x48
213         0x6161a3c2, 0x35355f6a, 0x5757f9ae, 0xb9b9d069, 0x86
214         0xe1e138d9, 0xf8f813eb, 0x9898b32b, 0x11113322, 0x69
215         0x9b9bb62d, 0x1e1e223c, 0x87879215, 0xe9e920c9, 0xce
216         0x8c8c8f03, 0xa1a1f859, 0x89898009, 0x0d0d171a, 0xbf
217         0x4141c382, 0x9999b029, 0x2d2d775a, 0xf0f0111e, 0xb0
218     }
219
220     // Lookup tables for decryption.
221     // These can be recomputed by adapting the tests in aes_test
222
223     var td0 = [256]uint32{
224         0x51f4a750, 0x7e416553, 0x1a17a4c3, 0x3a275e96, 0x3b
225         0x2030fa55, 0xad766df6, 0x88cc7691, 0xf5024c25, 0x4f
226         0xdeb15a49, 0x25ba1b67, 0x45ea0e98, 0x5dfec0e1, 0xc3
227         0x038f5fe7, 0x15929c95, 0xbf6d7aeb, 0x955259da, 0xd4
228         0x75c2896a, 0xf48e7978, 0x99583e6b, 0x27b971dd, 0xbe
229         0x63df4a18, 0xe51a3182, 0x97513360, 0x62537f45, 0xb1
230         0x70486858, 0x8f45fd19, 0x94de6c87, 0x527bf8b7, 0xab
231         0xb2eb2807, 0x2fb5c203, 0x86c57b9a, 0xd33708a5, 0x30
232         0x8acf1c2b, 0xa779b492, 0xf307f2f0, 0x4e69e2a1, 0x65
233         0x342e539d, 0xa2f355a0, 0x058ae132, 0xa4f6eb75, 0x0b
234         0x3e218af9, 0x96dd063d, 0xdd3e05ae, 0x4de6bd46, 0x91
235         0x1998fb24, 0xd6bde997, 0x894043cc, 0x67d99e77, 0xb0
236         0xa17c0a47, 0x7c420fe9, 0xf8841ec9, 0x00000000, 0x09
237         0xfd0efffb, 0x0f853856, 0x3daed51e, 0x362d3927, 0x0a
238         0x0c0a67b1, 0x9357e70f, 0xb4ee96d2, 0x1b9b919e, 0x80
239         0xe293ba0a, 0xc0a02ae5, 0x3c22e043, 0x121b171d, 0x0e

```

```

240      0x57f11985, 0xaf75074c, 0xee99dabb, 0xa37f60fd, 0xf7
241      0x8b432976, 0xcb23c6dc, 0xb6edfc68, 0xb8e4f163, 0xd7
242      0x854a247d, 0xd2bb3df8, 0xae93211, 0xc729a16d, 0x1d
243      0x2bb3166c, 0xa970b999, 0x119448fa, 0x47e96422, 0xa8
244      0x87494ec7, 0xd938d1c1, 0x8ccaa2fe, 0x98d40b36, 0xa6
245      0x2c3a9de4, 0x5078920d, 0x6a5fcc9b, 0x547e4662, 0xf6
246      0x9f5d80be, 0x69d0937c, 0x6fd52da9, 0xcf2512b3, 0xc8
247      0xcd267809, 0x6e5918f4, 0xec9ab701, 0x834f9aa8, 0xe6
248      0xbae79bd9, 0x4a6f36ce, 0xea9f09d4, 0x29b07cd6, 0x31
249      0x744ebc37, 0xfc82caa6, 0xe090d0b0, 0x33a7d815, 0xf1
250      0x764dd68d, 0x43efb04d, 0xccaa4d54, 0xe49604df, 0x9e
251      0x9d5eea04, 0x018c355d, 0xfa877473, 0xfb0b412e, 0xb3
252      0x9ad7618c, 0x37a10c7a, 0x59f8148e, 0xeb133c89, 0xce
253      0x9cd2df59, 0x55f2733f, 0x1814ce79, 0x73c737bf, 0x53
254      0xcaaff381, 0xb968c43e, 0x3824342c, 0xc2a3405f, 0x16
255      0x39a80171, 0x080cb3de, 0xd8b4e49c, 0x6456c190, 0x7b
256  }
257  var td1 = [256]uint32{
258      0x5051f4a7, 0x537e4165, 0xc31a17a4, 0x963a275e, 0xcb
259      0x552030fa, 0xf6ad766d, 0x9188cc76, 0x25f5024c, 0xfc
260      0x49deb15a, 0x6725ba1b, 0x9845ea0e, 0xe15dfec0, 0x02
261      0xe7038f5f, 0x9515929c, 0xebbf6d7a, 0xda955259, 0x2d
262      0x6a75c289, 0x78f48e79, 0x6b99583e, 0xdd27b971, 0xb6
263      0x1863df4a, 0x82e51a31, 0x60975133, 0x4562537f, 0xe0
264      0x58704868, 0x198f45fd, 0x8794de6c, 0xb7527bf8, 0x23
265      0x07b2eb28, 0x032fb5c2, 0x9a86c57b, 0xa5d33708, 0xf2
266      0x2b8acf1c, 0x92a779b4, 0xf0f307f2, 0xa14e69e2, 0xcd
267      0x9d342e53, 0xa0a2f355, 0x32058ae1, 0x75a4f6eb, 0x39
268      0xf93e218a, 0x3d96dd06, 0xaedd3e05, 0x464de6bd, 0xb5
269      0x241998fb, 0x97d6bde9, 0xcc894043, 0x7767d99e, 0xbd
270      0x47a17c0a, 0xe97c420f, 0xc9f8841e, 0x00000000, 0x83
271      0xfbfd0eff, 0x560f8538, 0x1e3daed5, 0x27362d39, 0x64
272      0xb10c0a67, 0x0f9357e7, 0xd2b4ee96, 0x9e1b9b91, 0x4f
273      0x0ae293ba, 0xe5c0a02a, 0x433c22e0, 0x1d121b17, 0x0b
274      0x8557f119, 0x4caf7507, 0xbbee99dd, 0xfda37f60, 0x9f
275      0x768b4329, 0xdcdb23c6, 0x68b6edfc, 0x63b8e4f1, 0xca
276      0x7d854a24, 0xf8d2bb3d, 0x11aef932, 0x6dc729a1, 0x4b
277      0x6c2bb316, 0x99a970b9, 0xfa119448, 0x2247e964, 0xc4
278      0xc787494e, 0xc1d938d1, 0xfe8ccaa2, 0x3698d40b, 0xcf
279      0xe42c3a9d, 0x0d507892, 0x9b6a5fcc, 0x62547e46, 0xc2
280      0xbe9f5d80, 0x7c69d093, 0xa96fd52d, 0xb3cf2512, 0x3b
281      0x09cd2678, 0xf46e5918, 0x01ec9ab7, 0xa8834f9a, 0x65
282      0xd9bae79b, 0xce4a6f36, 0xd4ea9f09, 0xd629b07c, 0xaf
283      0x37744ebc, 0xa6fc82ca, 0xb0e090d0, 0x1533a7d8, 0x4a
284      0x8d764dd6, 0x4d43efb0, 0x54ccaa4d, 0xdfe49604, 0xe3
285      0x049d5eea, 0x5d018c35, 0x73fa8774, 0x2efb0b41, 0x5a
286      0x8c9ad761, 0x7a37a10c, 0x8e59f814, 0x89eb133c, 0xee
287      0x599cd2df, 0x3f55f273, 0x791814ce, 0xbf73c737, 0xea
288      0x81caaff3, 0x3eb968c4, 0x2c382434, 0x5fc2a340, 0x72

```

```

289         0x7139a801, 0xde080cb3, 0x9cd8b4e4, 0x906456c1, 0x61
290     }
291     var td2 = [256]uint32{
292         0xa75051f4, 0x65537e41, 0xa4c31a17, 0x5e963a27, 0x6b
293         0xfa552030, 0x6df6ad76, 0x769188cc, 0x4c25f502, 0xd7
294         0x5a49deb1, 0x1b6725ba, 0x0e9845ea, 0xc0e15dfe, 0x75
295         0x5fe7038f, 0x9c951592, 0x7aebbf6d, 0x59da9552, 0x83
296         0x896a75c2, 0x7978f48e, 0x3e6b9958, 0x71dd27b9, 0x4f
297         0x4a1863df, 0x3182e51a, 0x33609751, 0x7f456253, 0x77
298         0x68587048, 0xfd198f45, 0x6c8794de, 0xf8b7527b, 0xd3
299         0x2807b2eb, 0xc2032fb5, 0x7b9a86c5, 0x08a5d337, 0x87
300         0x1c2b8acf, 0xb492a779, 0xf2f0f307, 0xe2a14e69, 0xf4
301         0x539d342e, 0x55a0a2f3, 0xe132058a, 0xeb75a4f6, 0xec
302         0x8af93e21, 0x063d96dd, 0x05aedd3e, 0xbd464de6, 0x8d
303         0xfb241998, 0xe997d6bd, 0x43cc8940, 0x9e7767d9, 0x42
304         0x0a47a17c, 0x0fe97c42, 0x1ec9f884, 0x00000000, 0x86
305         0xfffbfd0e, 0x38560f85, 0xd51e3dae, 0x3927362d, 0xd9
306         0x67b10c0a, 0xe70f9357, 0x96d2b4ee, 0x919e1b9b, 0xc5
307         0xba0ae293, 0x2ae5c0a0, 0xe0433c22, 0x171d121b, 0x0d
308         0x198557f1, 0x074caf75, 0xddbbe99, 0x60fda37f, 0x26
309         0x29768b43, 0xc6dcc23, 0xfc68b6ed, 0xf163b8e4, 0xdc
310         0x247d854a, 0x3df8d2bb, 0x3211aef9, 0xa16dc729, 0x2f
311         0x166c2bb3, 0xb999a970, 0x48fa1194, 0x642247e9, 0x8c
312         0x4ec78749, 0xd1c1d938, 0xa2fe8cca, 0x0b3698d4, 0x81
313         0x9de42c3a, 0x920d5078, 0xcc9b6a5f, 0x4662547e, 0x13
314         0x80be9f5d, 0x937c69d0, 0x2da96fd5, 0x12b3cf25, 0x99
315         0x7809cd26, 0x18f46e59, 0xb701ec9a, 0x9aa8834f, 0x6e
316         0x9bd9bae7, 0x36ce4a6f, 0x09d4ea9f, 0x7cd629b0, 0xb2
317         0xbc37744e, 0xcaa6fc82, 0xd0b0e090, 0xd81533a7, 0x98
318         0xd68d764d, 0xb04d43ef, 0x4d54ccaa, 0x04dfe496, 0xb5
319         0xea049d5e, 0x355d018c, 0x7473fa87, 0x412efb0b, 0x1d
320         0x618c9ad7, 0x0c7a37a1, 0x148e59f8, 0x3c89eb13, 0x27
321         0xdf599cd2, 0x733f55f2, 0xce791814, 0x37bf73c7, 0xcd
322         0xf381caaf, 0xc43eb968, 0x342c3824, 0x405fc2a3, 0xc3
323         0x017139a8, 0xb3de080c, 0xe49cd8b4, 0xc1906456, 0x84
324     }
325     var td3 = [256]uint32{
326         0xf4a75051, 0x4165537e, 0x17a4c31a, 0x275e963a, 0xab
327         0x30fa5520, 0x766df6ad, 0xcc769188, 0x024c25f5, 0xe5
328         0xb15a49de, 0xba1b6725, 0xea0e9845, 0xfec0e15d, 0x2f
329         0x8f5fe703, 0x929c9515, 0x6d7aebbf, 0x5259da95, 0xbe
330         0xc2896a75, 0x8e7978f4, 0x583e6b99, 0xb971dd27, 0xe1
331         0xdf4a1863, 0x1a3182e5, 0x51336097, 0x537f4562, 0x64
332         0x48685870, 0x45fd198f, 0xde6c8794, 0x7bf8b752, 0x73
333         0xeb2807b2, 0xb5c2032f, 0xc57b9a86, 0x3708a5d3, 0x28
334         0xcf1c2b8a, 0x79b492a7, 0x07f2f0f3, 0x69e2a14e, 0xda
335         0x2e539d34, 0xf355a0a2, 0x8ae13205, 0xf6eb75a4, 0x83
336         0x218af93e, 0xdd063d96, 0x3e05aedd, 0xe6bd464d, 0x54
337         0x98fb2419, 0xbde997d6, 0x4043cc89, 0xd99e7767, 0xe8

```

```
338      0x7c0a47a1, 0x420fe97c, 0x841ec9f8, 0x00000000, 0x80
339      0x0efffbfd, 0x8538560f, 0xaed51e3d, 0x2d392736, 0x0f
340      0x0a67b10c, 0x57e70f93, 0xee96d2b4, 0x9b919e1b, 0xc0
341      0x93ba0ae2, 0xa02ae5c0, 0x22e0433c, 0x1b171d12, 0x09
342      0xf1198557, 0x75074caf, 0x99ddbbee, 0x7f60fda3, 0x01
343      0x4329768b, 0x23c6dccb, 0xedfc68b6, 0xe4f163b8, 0x31
344      0x4a247d85, 0xbb3df8d2, 0xf93211ae, 0x29a16dc7, 0x9e
345      0xb3166c2b, 0x70b999a9, 0x9448fa11, 0xe9642247, 0xfc
346      0x494ec787, 0x38d1c1d9, 0xcaa2fe8c, 0xd40b3698, 0xf5
347      0x3a9de42c, 0x78920d50, 0x5fcc9b6a, 0x7e466254, 0x8d
348      0x5d80be9f, 0xd0937c69, 0xd52da96f, 0x2512b3cf, 0xac
349      0x267809cd, 0x5918f46e, 0x9ab701ec, 0x4f9aa883, 0x95
350      0xe79bd9ba, 0x6f36ce4a, 0x9f09d4ea, 0xb07cd629, 0xa4
351      0x4ebc3774, 0x82caa6fc, 0x90d0b0e0, 0xa7d81533, 0x04
352      0x4dd68d76, 0xefb04d43, 0xaa4d54cc, 0x9604dfe4, 0xd1
353      0x5eea049d, 0x8c355d01, 0x877473fa, 0x0b412efb, 0x67
354      0xd7618c9a, 0xa10c7a37, 0xf8148e59, 0x133c89eb, 0xa9
355      0xd2df599c, 0xf2733f55, 0x14ce7918, 0xc737bf73, 0xf7
356      0xaff381ca, 0x68c43eb9, 0x24342c38, 0xa3405fc2, 0x1d
357      0xa8017139, 0x0cb3de08, 0xb4e49cd8, 0x56c19064, 0xcb
358 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/cipher/cbc.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Cipher block chaining (CBC) mode.
6
7 // CBC provides confidentiality by xoring (chaining) each pl
8 // with the previous ciphertext block before applying the bl
9
10 // See NIST SP 800-38A, pp 10-11
11
12 package cipher
13
14 type cbc struct {
15     b      Block
16     blockSize int
17     iv     []byte
18     tmp    []byte
19 }
20
21 func newCBC(b Block, iv []byte) *cbc {
22     return &cbc{
23         b:      b,
24         blockSize: b.BlockSize(),
25         iv:     dup(iv),
26         tmp:    make([]byte, b.BlockSize()),
27     }
28 }
29
30 type cbcEncrypter cbc
31
32 // NewCBCEncrypter returns a BlockMode which encrypts in cip
33 // mode, using the given Block. The length of iv must be the
34 // Block's block size.
35 func NewCBCEncrypter(b Block, iv []byte) BlockMode {
36     return (*cbcEncrypter)(newCBC(b, iv))
37 }
38
39 func (x *cbcEncrypter) BlockSize() int { return x.blockSize
40
41 func (x *cbcEncrypter) CryptBlocks(dst, src []byte) {
```

```

42         for len(src) > 0 {
43             for i := 0; i < x.blockSize; i++ {
44                 x.iv[i] ^= src[i]
45             }
46             x.b.Encrypt(x.iv, x.iv)
47             for i := 0; i < x.blockSize; i++ {
48                 dst[i] = x.iv[i]
49             }
50             src = src[x.blockSize:]
51             dst = dst[x.blockSize:]
52         }
53     }
54
55     type cbcDecrypter cbc
56
57     // NewCBCDecrypter returns a BlockMode which decrypts in cip
58     // mode, using the given Block. The length of iv must be the
59     // Block's block size and must match the iv used to encrypt
60     func NewCBCDecrypter(b Block, iv []byte) BlockMode {
61         return (*cbcDecrypter)(newCBC(b, iv))
62     }
63
64     func (x *cbcDecrypter) BlockSize() int { return x.blockSize
65
66     func (x *cbcDecrypter) CryptBlocks(dst, src []byte) {
67         for len(src) > 0 {
68             x.b.Decrypt(x.tmp, src[:x.blockSize])
69             for i := 0; i < x.blockSize; i++ {
70                 x.tmp[i] ^= x.iv[i]
71                 x.iv[i] = src[i]
72                 dst[i] = x.tmp[i]
73             }
74
75             src = src[x.blockSize:]
76             dst = dst[x.blockSize:]
77         }
78     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/cipher/cfb.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // CFB (Cipher Feedback) Mode.
6
7 package cipher
8
9 type cfb struct {
10     b      Block
11     out    []byte
12     outUsed int
13     decrypt bool
14 }
15
16 // NewCFBEncrypter returns a Stream which encrypts with ciph
17 // using the given Block. The iv must be the same length as
18 // size.
19 func NewCFBEncrypter(block Block, iv []byte) Stream {
20     return newCFB(block, iv, false)
21 }
22
23 // NewCFBDecrypter returns a Stream which decrypts with ciph
24 // using the given Block. The iv must be the same length as
25 // size.
26 func NewCFBDecrypter(block Block, iv []byte) Stream {
27     return newCFB(block, iv, true)
28 }
29
30 func newCFB(block Block, iv []byte, decrypt bool) Stream {
31     blockSize := block.BlockSize()
32     if len(iv) != blockSize {
33         return nil
34     }
35
36     x := &cfb{
37         b:      block,
38         out:    make([]byte, blockSize),
39         outUsed: 0,
40         decrypt: decrypt,
41     }
```

```

42         block.Encrypt(x.out, iv)
43
44         return x
45     }
46
47     func (x *cfb) XORKeyStream(dst, src []byte) {
48         for i := 0; i < len(src); i++ {
49             if x.outUsed == len(x.out) {
50                 x.b.Encrypt(x.out, x.out)
51                 x.outUsed = 0
52             }
53
54             if x.decrypt {
55                 t := src[i]
56                 dst[i] = src[i] ^ x.out[x.outUsed]
57                 x.out[x.outUsed] = t
58             } else {
59                 x.out[x.outUsed] ^= src[i]
60                 dst[i] = x.out[x.outUsed]
61             }
62             x.outUsed++
63         }
64     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/cipher/cipher.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package cipher implements standard block cipher modes tha
6 // around low-level block cipher implementations.
7 // See http://csrc.nist.gov/groups/ST/toolkit/BCM/current\_mo
8 // and NIST Special Publication 800-38A.
9 package cipher
10
11 // A Block represents an implementation of block cipher
12 // using a given key. It provides the capability to encrypt
13 // or decrypt individual blocks. The mode implementations
14 // extend that capability to streams of blocks.
15 type Block interface {
16     // BlockSize returns the cipher's block size.
17     BlockSize() int
18
19     // Encrypt encrypts the first block in src into dst.
20     // Dst and src may point at the same memory.
21     Encrypt(dst, src []byte)
22
23     // Decrypt decrypts the first block in src into dst.
24     // Dst and src may point at the same memory.
25     Decrypt(dst, src []byte)
26 }
27
28 // A Stream represents a stream cipher.
29 type Stream interface {
30     // XORKeyStream XORs each byte in the given slice wi
31     // cipher's key stream. Dst and src may point to the
32     XORKeyStream(dst, src []byte)
33 }
34
35 // A BlockMode represents a block cipher running in a block-
36 // ECB etc).
37 type BlockMode interface {
38     // BlockSize returns the mode's block size.
39     BlockSize() int
40
41     // CryptBlocks encrypts or decrypts a number of bloc
```

```

42         // src must be a multiple of the block size. Dst and
43         // the same memory.
44         CryptBlocks(dst, src []byte)
45     }
46
47     // Utility routines
48
49     func shift1(dst, src []byte) byte {
50         var b byte
51         for i := len(src) - 1; i >= 0; i-- {
52             bb := src[i] >> 7
53             dst[i] = src[i]<<1 | b
54             b = bb
55         }
56         return b
57     }
58
59     func dup(p []byte) []byte {
60         q := make([]byte, len(p))
61         copy(q, p)
62         return q
63     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/cipher/ctr.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Counter (CTR) mode.
6
7 // CTR converts a block cipher into a stream cipher by
8 // repeatedly encrypting an incrementing counter and
9 // xoring the resulting stream of data with the input.
10
11 // See NIST SP 800-38A, pp 13-15
12
13 package cipher
14
15 type ctr struct {
16     b      Block
17     ctr    []byte
18     out    []byte
19     outUsed int
20 }
21
22 // NewCTR returns a Stream which encrypts/decrypts using the
23 // counter mode. The length of iv must be the same as the B
24 func NewCTR(block Block, iv []byte) Stream {
25     if len(iv) != block.BlockSize() {
26         panic("cipher.NewCTR: iv length must equal b
27     }
28
29     return &ctr{
30         b:      block,
31         ctr:    dup(iv),
32         out:    make([]byte, len(iv)),
33         outUsed: len(iv),
34     }
35 }
36
37 func (x *ctr) XORKeyStream(dst, src []byte) {
38     for i := 0; i < len(src); i++ {
39         if x.outUsed == len(x.ctr) {
40             x.b.Encrypt(x.out, x.ctr)
41             x.outUsed = 0
```

```
42
43         // Increment counter
44         for i := len(x.ctr) - 1; i >= 0; i--
45             x.ctr[i]++
46             if x.ctr[i] != 0 {
47                 break
48             }
49         }
50     }
51
52     dst[i] = src[i] ^ x.out[x.outUsed]
53     x.outUsed++
54 }
55 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/cipher/io.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cipher
6
7 import "io"
8
9 // The Stream* objects are so simple that all their members
10 // can create them themselves.
11
12 // StreamReader wraps a Stream into an io.Reader. It calls X
13 // to process each slice of data which passes through.
14 type StreamReader struct {
15     S Stream
16     R io.Reader
17 }
18
19 func (r StreamReader) Read(dst []byte) (n int, err error) {
20     n, err = r.R.Read(dst)
21     r.S.XORKeyStream(dst[:n], dst[:n])
22     return
23 }
24
25 // StreamWriter wraps a Stream into an io.Writer. It calls X
26 // to process each slice of data which passes through. If an
27 // returns short then the StreamWriter is out of sync and mu
28 type StreamWriter struct {
29     S Stream
30     W io.Writer
31     Err error
32 }
33
34 func (w StreamWriter) Write(src []byte) (n int, err error) {
35     if w.Err != nil {
36         return 0, w.Err
37     }
38     c := make([]byte, len(src))
39     w.S.XORKeyStream(c, src)
40     n, err = w.W.Write(c)
41     if n != len(src) {
```

```
42             if err == nil { // should never happen
43                 err = io.ErrShortWrite
44             }
45             w.Err = err
46         }
47         return
48     }
49
50     func (w StreamWriter) Close() error {
51         // This saves us from either requiring a WriteCloser
52         // StreamWriterCloser.
53         return w.W.(io.Closer).Close()
54     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/cipher/ofb.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // OFB (Output Feedback) Mode.
6
7 package cipher
8
9 type ofb struct {
10     b      Block
11     out    []byte
12     outUsed int
13 }
14
15 // NewOFB returns a Stream that encrypts or decrypts using t
16 // in output feedback mode. The initialization vector iv's l
17 // to b's block size.
18 func NewOFB(b Block, iv []byte) Stream {
19     blockSize := b.BlockSize()
20     if len(iv) != blockSize {
21         return nil
22     }
23
24     x := &ofb{
25         b:      b,
26         out:    make([]byte, blockSize),
27         outUsed: 0,
28     }
29     b.Encrypt(x.out, iv)
30
31     return x
32 }
33
34 func (x *ofb) XORKeyStream(dst, src []byte) {
35     for i, s := range src {
36         if x.outUsed == len(x.out) {
37             x.b.Encrypt(x.out, x.out)
38             x.outUsed = 0
39         }
40
41         dst[i] = s ^ x.out[x.outUsed]
```

```
42         x.outUsed++
43     }
44 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/des/block.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package des
6
7 import (
8     "encoding/binary"
9 )
10
11 func cryptBlock(subkeys []uint64, dst, src []byte, decrypt b
12     b := binary.BigEndian.Uint64(src)
13     b = permuteBlock(b, initialPermutation[:])
14     left, right := uint32(b>>32), uint32(b)
15
16     var subkey uint64
17     for i := 0; i < 16; i++ {
18         if decrypt {
19             subkey = subkeys[15-i]
20         } else {
21             subkey = subkeys[i]
22         }
23
24         left, right = right, left^feistel(right, sub
25     }
26     // switch left & right and perform final permutation
27     preOutput := (uint64(right) << 32) | uint64(left)
28     binary.BigEndian.PutUint64(dst, permuteBlock(preOutp
29 }
30
31 // Encrypt one block from src into dst, using the subkeys.
32 func encryptBlock(subkeys []uint64, dst, src []byte) {
33     cryptBlock(subkeys, dst, src, false)
34 }
35
36 // Decrypt one block from src into dst, using the subkeys.
37 func decryptBlock(subkeys []uint64, dst, src []byte) {
38     cryptBlock(subkeys, dst, src, true)
39 }
40
41 // DES Feistel function
```

```

42 func feistel(right uint32, key uint64) (result uint32) {
43     sBoxLocations := key ^ permuteBlock(uint64(right), e
44     var sBoxResult uint32
45     for i := uint8(0); i < 8; i++ {
46         sBoxLocation := uint8(sBoxLocations>>42) & 0
47         sBoxLocations <=<= 6
48         // row determined by 1st and 6th bit
49         row := (sBoxLocation & 0x1) | ((sBoxLocation
50         // column is middle four bits
51         column := (sBoxLocation >> 1) & 0xf
52         sBoxResult |= uint32(sBoxes[i][row][column])
53     }
54     return uint32(permuteBlock(uint64(sBoxResult), permu
55 }
56
57 // general purpose function to perform DES block permutation
58 func permuteBlock(src uint64, permutation []uint8) (block ui
59     for position, n := range permutation {
60         bit := (src >> n) & 1
61         block |= bit << uint((len(permutation)-1)-po
62     }
63     return
64 }
65
66 // creates 16 28-bit blocks rotated according
67 // to the rotation schedule
68 func ksRotate(in uint32) (out []uint32) {
69     out = make([]uint32, 16)
70     last := in
71     for i := 0; i < 16; i++ {
72         // 28-bit circular left shift
73         left := (last << (4 + ksRotations[i])) >> 4
74         right := (last << 4) >> (32 - ksRotations[i]
75         out[i] = left | right
76         last = out[i]
77     }
78     return
79 }
80
81 // creates 16 56-bit subkeys from the original key
82 func (c *desCipher) generateSubkeys(keyBytes []byte) {
83     // apply PC1 permutation to key
84     key := binary.BigEndian.Uint64(keyBytes)
85     permutedKey := permuteBlock(key, permutedChoice1[:])
86
87     // rotate halves of permuted key according to the ro
88     leftRotations := ksRotate(uint32(permutedKey >> 28))
89     rightRotations := ksRotate(uint32(permutedKey<<4) >>
90
91     // generate subkeys

```

```
92         for i := 0; i < 16; i++ {
93             // combine halves to form 56-bit input to PC
94             pc2Input := uint64(leftRotations[i])<<28 | u
95             // apply PC2 permutation to 7 byte input
96             c.subkeys[i] = permuteBlock(pc2Input, permut
97         }
98     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/des/cipher.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package des
6
7 import (
8     "crypto/cipher"
9     "strconv"
10 )
11
12 // The DES block size in bytes.
13 const BlockSize = 8
14
15 type KeySizeError int
16
17 func (k KeySizeError) Error() string {
18     return "crypto/des: invalid key size " + strconv.Itoa(
19 )
20 }
21 // desCipher is an instance of DES encryption.
22 type desCipher struct {
23     subkeys [16]uint64
24 }
25
26 // NewCipher creates and returns a new cipher.Block.
27 func NewCipher(key []byte) (cipher.Block, error) {
28     if len(key) != 8 {
29         return nil, KeySizeError(len(key))
30     }
31
32     c := new(desCipher)
33     c.generateSubkeys(key)
34     return c, nil
35 }
36
37 func (c *desCipher) BlockSize() int { return BlockSize }
38
39 func (c *desCipher) Encrypt(dst, src []byte) { encryptBlock(
40
41 func (c *desCipher) Decrypt(dst, src []byte) { decryptBlock(
```

```

42
43 // A tripleDESCipher is an instance of TripleDES encryption.
44 type tripleDESCipher struct {
45     cipher1, cipher2, cipher3 desCipher
46 }
47
48 // NewTripleDESCipher creates and returns a new cipher.Block
49 func NewTripleDESCipher(key []byte) (cipher.Block, error) {
50     if len(key) != 24 {
51         return nil, KeySizeError(len(key))
52     }
53
54     c := new(tripleDESCipher)
55     c.cipher1.generateSubkeys(key[:8])
56     c.cipher2.generateSubkeys(key[8:16])
57     c.cipher3.generateSubkeys(key[16:])
58     return c, nil
59 }
60
61 func (c *tripleDESCipher) BlockSize() int { return BlockSize
62
63 func (c *tripleDESCipher) Encrypt(dst, src []byte) {
64     c.cipher1.Encrypt(dst, src)
65     c.cipher2.Decrypt(dst, dst)
66     c.cipher3.Encrypt(dst, dst)
67 }
68
69 func (c *tripleDESCipher) Decrypt(dst, src []byte) {
70     c.cipher3.Decrypt(dst, src)
71     c.cipher2.Encrypt(dst, dst)
72     c.cipher1.Decrypt(dst, dst)
73 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/des/const.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package des implements the Data Encryption Standard (DES)
6 // Triple Data Encryption Algorithm (TDEA) as defined
7 // in U.S. Federal Information Processing Standards Publicat
8 package des
9
10 // Used to perform an initial permutation of a 64-bit input
11 var initialPermutation = [64]byte{
12     6, 14, 22, 30, 38, 46, 54, 62,
13     4, 12, 20, 28, 36, 44, 52, 60,
14     2, 10, 18, 26, 34, 42, 50, 58,
15     0, 8, 16, 24, 32, 40, 48, 56,
16     7, 15, 23, 31, 39, 47, 55, 63,
17     5, 13, 21, 29, 37, 45, 53, 61,
18     3, 11, 19, 27, 35, 43, 51, 59,
19     1, 9, 17, 25, 33, 41, 49, 57,
20 }
21
22 // Used to perform a final permutation of a 4-bit preoutput
23 // inverse of initialPermutation
24 var finalPermutation = [64]byte{
25     24, 56, 16, 48, 8, 40, 0, 32,
26     25, 57, 17, 49, 9, 41, 1, 33,
27     26, 58, 18, 50, 10, 42, 2, 34,
28     27, 59, 19, 51, 11, 43, 3, 35,
29     28, 60, 20, 52, 12, 44, 4, 36,
30     29, 61, 21, 53, 13, 45, 5, 37,
31     30, 62, 22, 54, 14, 46, 6, 38,
32     31, 63, 23, 55, 15, 47, 7, 39,
33 }
34
35 // Used to expand an input block of 32 bits, producing an ou
36 // bits.
37 var expansionFunction = [48]byte{
38     0, 31, 30, 29, 28, 27, 28, 27,
39     26, 25, 24, 23, 24, 23, 22, 21,
40     20, 19, 20, 19, 18, 17, 16, 15,
41     16, 15, 14, 13, 12, 11, 12, 11,
```

```

42         10, 9, 8, 7, 8, 7, 6, 5,
43         4, 3, 4, 3, 2, 1, 0, 31,
44     }
45
46 // Yields a 32-bit output from a 32-bit input
47 var permutationFunction = [32]byte{
48     16, 25, 12, 11, 3, 20, 4, 15,
49     31, 17, 9, 6, 27, 14, 1, 22,
50     30, 24, 8, 18, 0, 5, 29, 23,
51     13, 19, 2, 26, 10, 21, 28, 7,
52 }
53
54 // Used in the key schedule to select 56 bits
55 // from a 64-bit input.
56 var permutedChoice1 = [56]byte{
57     7, 15, 23, 31, 39, 47, 55, 63,
58     6, 14, 22, 30, 38, 46, 54, 62,
59     5, 13, 21, 29, 37, 45, 53, 61,
60     4, 12, 20, 28, 1, 9, 17, 25,
61     33, 41, 49, 57, 2, 10, 18, 26,
62     34, 42, 50, 58, 3, 11, 19, 27,
63     35, 43, 51, 59, 36, 44, 52, 60,
64 }
65
66 // Used in the key schedule to produce each subkey by select
67 // the 56-bit input
68 var permutedChoice2 = [48]byte{
69     42, 39, 45, 32, 55, 51, 53, 28,
70     41, 50, 35, 46, 33, 37, 44, 52,
71     30, 48, 40, 49, 29, 36, 43, 54,
72     15, 4, 25, 19, 9, 1, 26, 16,
73     5, 11, 23, 8, 12, 7, 17, 0,
74     22, 3, 10, 14, 6, 20, 27, 24,
75 }
76
77 // 8 S-boxes composed of 4 rows and 16 columns
78 // Used in the DES cipher function
79 var sBoxes = [8][4][16]uint8{
80     // S-box 1
81     {
82         {14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5
83         {0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9
84         {4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3,
85         {15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10,
86     },
87     // S-box 2
88     {
89         {15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12,
90         {3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6,
91         {0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9,

```

```

92         {13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0,
93     },
94     // S-box 3
95     {
96         {10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11
97         {13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12,
98         {13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5,
99         {1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11,
100    },
101    // S-box 4
102    {
103        {7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11,
104        {13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1,
105        {10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5
106        {3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12,
107    },
108    // S-box 5
109    {
110        {2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13,
111        {14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3
112        {4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6,
113        {11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10
114    },
115    // S-box 6
116    {
117        {12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14,
118        {10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0,
119        {9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1,
120        {4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6,
121    },
122    // S-box 7
123    {
124        {4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5,
125        {13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2,
126        {1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0
127        {6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14,
128    },
129    // S-box 8
130    {
131        {13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5,
132        {1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0,
133        {7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15
134        {2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3,
135    },
136 }
137
138 // Size of left rotation per round in each half of the key s
139 var ksRotations = [16]uint8{1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2,

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/crypto/dsa/dsa.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package dsa implements the Digital Signature Algorithm, a
6 package dsa
7
8 import (
9     "errors"
10    "io"
11    "math/big"
12 )
13
14 // Parameters represents the domain parameters for a key. Th
15 // be shared across many keys. The bit length of Q must be a
16 type Parameters struct {
17     P, Q, G *big.Int
18 }
19
20 // PublicKey represents a DSA public key.
21 type PublicKey struct {
22     Parameters
23     Y *big.Int
24 }
25
26 // PrivateKey represents a DSA private key.
27 type PrivateKey struct {
28     PublicKey
29     X *big.Int
30 }
31
32 // ErrInvalidPublicKey results when a public key is not usab
33 // FIPS is quite strict about the format of DSA keys, but ot
34 // less so. Thus, when using keys which may have been genera
35 // this error must be handled.
36 var ErrInvalidPublicKey = errors.New("crypto/dsa: invalid pu
37
38 // ParameterSizes is a enumeration of the acceptable bit len
39 // in a set of DSA parameters. See FIPS 186-3, section 4.2.
40 type ParameterSizes int
41
42 const (
43     L1024N160 ParameterSizes = iota
44     L2048N224
```

```

45         L2048N256
46         L3072N256
47     )
48
49     // numMRTests is the number of Miller-Rabin primality tests
50     // pick the largest recommended number from table C.1 of FIP
51     const numMRTests = 64
52
53     // GenerateParameters puts a random, valid set of DSA parame
54     // This function takes many seconds, even on fast machines.
55     func GenerateParameters(params *Parameters, rand io.Reader,
56         // This function doesn't follow FIPS 186-3 exactly i
57         // use a verification seed to generate the primes. T
58         // seed doesn't appear to be exported or used by oth
59         // omitting it makes the code cleaner.
60
61         var L, N int
62         switch sizes {
63         case L1024N160:
64             L = 1024
65             N = 160
66         case L2048N224:
67             L = 2048
68             N = 224
69         case L2048N256:
70             L = 2048
71             N = 256
72         case L3072N256:
73             L = 3072
74             N = 256
75         default:
76             return errors.New("crypto/dsa: invalid Param
77         }
78
79         qBytes := make([]byte, N/8)
80         pBytes := make([]byte, L/8)
81
82         q := new(big.Int)
83         p := new(big.Int)
84         rem := new(big.Int)
85         one := new(big.Int)
86         one.SetInt64(1)
87
88     GeneratePrimes:
89         for {
90             _, err = io.ReadFull(rand, qBytes)
91             if err != nil {
92                 return
93             }
94

```

```

95         qBytes[len(qBytes)-1] |= 1
96         qBytes[0] |= 0x80
97         q.SetBytes(qBytes)
98
99         if !q.ProbablyPrime(numMRTests) {
100             continue
101         }
102
103         for i := 0; i < 4*L; i++ {
104             _, err = io.ReadFull(rand, pBytes)
105             if err != nil {
106                 return
107             }
108
109             pBytes[len(pBytes)-1] |= 1
110             pBytes[0] |= 0x80
111
112             p.SetBytes(pBytes)
113             rem.Mod(p, q)
114             rem.Sub(rem, one)
115             p.Sub(p, rem)
116             if p.BitLen() < L {
117                 continue
118             }
119
120             if !p.ProbablyPrime(numMRTests) {
121                 continue
122             }
123
124             params.P = p
125             params.Q = q
126             break GeneratePrimes
127         }
128     }
129
130     h := new(big.Int)
131     h.SetInt64(2)
132     g := new(big.Int)
133
134     pm1 := new(big.Int).Sub(p, one)
135     e := new(big.Int).Div(pm1, q)
136
137     for {
138         g.Exp(h, e, p)
139         if g.Cmp(one) == 0 {
140             h.Add(h, one)
141             continue
142         }
143     }

```

```

144         params.G = g
145         return
146     }
147
148     panic("unreachable")
149 }
150
151 // GenerateKey generates a public&private key pair. The Para
152 // PrivateKey must already be valid (see GenerateParameters)
153 func GenerateKey(priv *PrivateKey, rand io.Reader) error {
154     if priv.P == nil || priv.Q == nil || priv.G == nil {
155         return errors.New("crypto/dsa: parameters no
156     }
157
158     x := new(big.Int)
159     xBytes := make([]byte, priv.Q.BitLen()/8)
160
161     for {
162         _, err := io.ReadFull(rand, xBytes)
163         if err != nil {
164             return err
165         }
166         x.SetBytes(xBytes)
167         if x.Sign() != 0 && x.Cmp(priv.Q) < 0 {
168             break
169         }
170     }
171
172     priv.X = x
173     priv.Y = new(big.Int)
174     priv.Y.Exp(priv.G, x, priv.P)
175     return nil
176 }
177
178 // Sign signs an arbitrary length hash (which should be the
179 // larger message) using the private key, priv. It returns t
180 // pair of integers. The security of the private key depends
181 // rand.
182 //
183 // Note that FIPS 186-3 section 4.6 specifies that the hash
184 // to the byte-length of the subgroup. This function does no
185 // truncation itself.
186 func Sign(rand io.Reader, priv *PrivateKey, hash []byte) (r,
187     // FIPS 186-3, section 4.6
188
189     n := priv.Q.BitLen()
190     if n&7 != 0 {
191         err = ErrInvalidPublicKey
192         return

```

```

193     }
194     n >>= 3
195
196     for {
197         k := new(big.Int)
198         buf := make([]byte, n)
199         for {
200             _, err = io.ReadFull(rand, buf)
201             if err != nil {
202                 return
203             }
204             k.SetBytes(buf)
205             if k.Sign() > 0 && k.Cmp(priv.Q) < 0
206                 break
207         }
208     }
209
210     kInv := new(big.Int).ModInverse(k, priv.Q)
211
212     r = new(big.Int).Exp(priv.G, k, priv.P)
213     r.Mod(r, priv.Q)
214
215     if r.Sign() == 0 {
216         continue
217     }
218
219     z := k.SetBytes(hash)
220
221     s = new(big.Int).Mul(priv.X, r)
222     s.Add(s, z)
223     s.Mod(s, priv.Q)
224     s.Mul(s, kInv)
225     s.Mod(s, priv.Q)
226
227     if s.Sign() != 0 {
228         break
229     }
230 }
231
232     return
233 }
234
235 // Verify verifies the signature in r, s of hash using the p
236 // reports whether the signature is valid.
237 //
238 // Note that FIPS 186-3 section 4.6 specifies that the hash
239 // to the byte-length of the subgroup. This function does no
240 // truncation itself.
241 func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool
242     // FIPS 186-3, section 4.7

```

```

243
244     if r.Sign() < 1 || r.Cmp(pub.Q) >= 0 {
245         return false
246     }
247     if s.Sign() < 1 || s.Cmp(pub.Q) >= 0 {
248         return false
249     }
250
251     w := new(big.Int).ModInverse(s, pub.Q)
252
253     n := pub.Q.BitLen()
254     if n&7 != 0 {
255         return false
256     }
257     z := new(big.Int).SetBytes(hash)
258
259     u1 := new(big.Int).Mul(z, w)
260     u1.Mod(u1, pub.Q)
261     u2 := w.Mul(r, w)
262     u2.Mod(u2, pub.Q)
263     v := u1.Exp(pub.G, u1, pub.P)
264     u2.Exp(pub.Y, u2, pub.P)
265     v.Mul(v, u2)
266     v.Mod(v, pub.P)
267     v.Mod(v, pub.Q)
268
269     return v.Cmp(r) == 0
270 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/ecdsa/ecdsa.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package ecdsa implements the Elliptic Curve Digital Signa
6 // defined in FIPS 186-3.
7 package ecdsa
8
9 // References:
10 // [NSA]: Suite B implementer's guide to FIPS 186-3,
11 //      http://www.nsa.gov/ia/_files/ecdsa.pdf
12 // [SECG]: SECG, SEC1
13 //      http://www.secg.org/download/aid-780/sec1-v2.pdf
14
15 import (
16     "crypto/elliptic"
17     "io"
18     "math/big"
19 )
20
21 // PublicKey represents an ECDSA public key.
22 type PublicKey struct {
23     elliptic.Curve
24     X, Y *big.Int
25 }
26
27 // PrivateKey represents a ECDSA private key.
28 type PrivateKey struct {
29     PublicKey
30     D *big.Int
31 }
32
33 var one = new(big.Int).SetInt64(1)
34
35 // randFieldElement returns a random element of the field un
36 // curve using the procedure given in [NSA] A.2.1.
37 func randFieldElement(c elliptic.Curve, rand io.Reader) (k *
38     params := c.Params()
39     b := make([]byte, params.BitSize/8+8)
40     _, err = io.ReadFull(rand, b)
41     if err != nil {
```

```

42         return
43     }
44
45     k = new(big.Int).SetBytes(b)
46     n := new(big.Int).Sub(params.N, one)
47     k.Mod(k, n)
48     k.Add(k, one)
49     return
50 }
51
52 // GenerateKey generates a public&private key pair.
53 func GenerateKey(c elliptic.Curve, rand io.Reader) (priv *Pr
54     k, err := randFieldElement(c, rand)
55     if err != nil {
56         return
57     }
58
59     priv = new(PrivateKey)
60     priv.PublicKey.Curve = c
61     priv.D = k
62     priv.PublicKey.X, priv.PublicKey.Y = c.ScalarBaseMul
63     return
64 }
65
66 // hashToInt converts a hash value to an integer. There is s
67 // about how this is done. [NSA] suggests that this is done
68 // manner, but [SECG] truncates the hash to the bit-length o
69 // first. We follow [SECG] because that's what OpenSSL does.
70 func hashToInt(hash []byte, c elliptic.Curve) *big.Int {
71     orderBits := c.Params().N.BitLen()
72     orderBytes := (orderBits + 7) / 8
73     if len(hash) > orderBytes {
74         hash = hash[:orderBytes]
75     }
76
77     ret := new(big.Int).SetBytes(hash)
78     excess := orderBytes*8 - orderBits
79     if excess > 0 {
80         ret.Rsh(ret, uint(excess))
81     }
82     return ret
83 }
84
85 // Sign signs an arbitrary length hash (which should be the
86 // larger message) using the private key, priv. It returns t
87 // pair of integers. The security of the private key depends
88 // rand.
89 func Sign(rand io.Reader, priv *PrivateKey, hash []byte) (r,
90     // See [NSA] 3.4.1
91     c := priv.PublicKey.Curve

```

```

92     N := c.Params().N
93
94     var k, kInv *big.Int
95     for {
96         for {
97             k, err = randFieldElement(c, rand)
98             if err != nil {
99                 r = nil
100                return
101            }
102
103            kInv = new(big.Int).ModInverse(k, N)
104            r, _ = priv.Curve.ScalarBaseMult(k.B
105            r.Mod(r, N)
106            if r.Sign() != 0 {
107                break
108            }
109        }
110
111        e := hashToInt(hash, c)
112        s = new(big.Int).Mul(priv.D, r)
113        s.Add(s, e)
114        s.Mul(s, kInv)
115        s.Mod(s, N)
116        if s.Sign() != 0 {
117            break
118        }
119    }
120
121    return
122 }
123
124 // Verify verifies the signature in r, s of hash using the p
125 // returns true iff the signature is valid.
126 func Verify(pub *PublicKey, hash []byte, r, s *big.Int) bool
127     // See [NSA] 3.4.2
128     c := pub.Curve
129     N := c.Params().N
130
131     if r.Sign() == 0 || s.Sign() == 0 {
132         return false
133     }
134     if r.Cmp(N) >= 0 || s.Cmp(N) >= 0 {
135         return false
136     }
137     e := hashToInt(hash, c)
138     w := new(big.Int).ModInverse(s, N)
139
140     u1 := e.Mul(e, w)

```

```
141         u2 := w.Mul(r, w)
142
143         x1, y1 := c.ScalarBaseMult(u1.Bytes())
144         x2, y2 := c.ScalarMult(pub.X, pub.Y, u2.Bytes())
145         if x1.Cmp(x2) == 0 {
146             return false
147         }
148         x, _ := c.Add(x1, y1, x2, y2)
149         x.Mod(x, N)
150         return x.Cmp(r) == 0
151     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/elliptic/elliptic.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package elliptic implements several standard elliptic cur
6 // fields.
7 package elliptic
8
9 // This package operates, internally, on Jacobian coordinate
10 // (x, y) position on the curve, the Jacobian coordinates ar
11 // where  $x = x1/z1^2$  and  $y = y1/z1^3$ . The greatest speedups co
12 // calculation can be performed within the transform (as in
13 // ScalarBaseMult). But even for Add and Double, it's faster
14 // reverse the transform than to operate in affine coordinat
15
16 import (
17     "io"
18     "math/big"
19     "sync"
20 )
21
22 // A Curve represents a short-form Weierstrass curve with a=
23 // See http://www.hyperelliptic.org/EFD/g1p/auto-shortw.html
24 type Curve interface {
25     // Params returns the parameters for the curve.
26     Params() *CurveParams
27     // IsOnCurve returns true if the given (x,y) lies on
28     IsOnCurve(x, y *big.Int) bool
29     // Add returns the sum of (x1,y1) and (x2,y2)
30     Add(x1, y1, x2, y2 *big.Int) (x, y *big.Int)
31     // Double returns 2*(x,y)
32     Double(x1, y1 *big.Int) (x, y *big.Int)
33     // ScalarMult returns k*(Bx,By) where k is a number
34     ScalarMult(x1, y1 *big.Int, scalar []byte) (x, y *bi
35     // ScalarBaseMult returns k*G, where G is the base p
36     // is an integer in big-endian form.
37     ScalarBaseMult(scalar []byte) (x, y *big.Int)
38 }
39
40 // CurveParams contains the parameters of an elliptic curve
41 // a generic, non-constant time implementation of Curve.
```

```

42 type CurveParams struct {
43     P      *big.Int // the order of the underlying field
44     N      *big.Int // the order of the base point
45     B      *big.Int // the constant of the curve equation
46     Gx, Gy *big.Int // (x,y) of the base point
47     BitSize int      // the size of the underlying field
48 }
49
50 func (curve *CurveParams) Params() *CurveParams {
51     return curve
52 }
53
54 func (curve *CurveParams) IsOnCurve(x, y *big.Int) bool {
55     //  $y^2 = x^3 - 3x + b$ 
56     y2 := new(big.Int).Mul(y, y)
57     y2.Mod(y2, curve.P)
58
59     x3 := new(big.Int).Mul(x, x)
60     x3.Mul(x3, x)
61
62     threeX := new(big.Int).Lsh(x, 1)
63     threeX.Add(threeX, x)
64
65     x3.Sub(x3, threeX)
66     x3.Add(x3, curve.B)
67     x3.Mod(x3, curve.P)
68
69     return x3.Cmp(y2) == 0
70 }
71
72 // affineFromJacobian reverses the Jacobian transform. See top of the file.
73 // top of the file.
74 func (curve *CurveParams) affineFromJacobian(x, y, z *big.Int) (*big.Int, *big.Int) {
75     zinv := new(big.Int).ModInverse(z, curve.P)
76     zinvsq := new(big.Int).Mul(zinv, zinv)
77
78     xOut = new(big.Int).Mul(x, zinvsq)
79     xOut.Mod(xOut, curve.P)
80     zinvsq.Mul(zinvsq, zinv)
81     yOut = new(big.Int).Mul(y, zinvsq)
82     yOut.Mod(yOut, curve.P)
83     return xOut, yOut
84 }
85
86 func (curve *CurveParams) Add(x1, y1, x2, y2 *big.Int) (*big.Int, *big.Int) {
87     z := new(big.Int).SetInt64(1)
88     return curve.affineFromJacobian(curve.addJacobian(x1, y1, x2, y2, z))
89 }
90
91 // addJacobian takes two points in Jacobian coordinates, (x1, y1, z1) and (x2, y2, z2)

```

```

92 // (x2, y2, z2) and returns their sum, also in Jacobian form
93 func (curve *CurveParams) addJacobian(x1, y1, z1, x2, y2, z2
94     // See http://hyperelliptic.org/EFD/g1p/auto-shortw-
95     z1z1 := new(big.Int).Mul(z1, z1)
96     z1z1.Mod(z1z1, curve.P)
97     z2z2 := new(big.Int).Mul(z2, z2)
98     z2z2.Mod(z2z2, curve.P)
99
100     u1 := new(big.Int).Mul(x1, z2z2)
101     u1.Mod(u1, curve.P)
102     u2 := new(big.Int).Mul(x2, z1z1)
103     u2.Mod(u2, curve.P)
104     h := new(big.Int).Sub(u2, u1)
105     if h.Sign() == -1 {
106         h.Add(h, curve.P)
107     }
108     i := new(big.Int).Lsh(h, 1)
109     i.Mul(i, i)
110     j := new(big.Int).Mul(h, i)
111
112     s1 := new(big.Int).Mul(y1, z2)
113     s1.Mul(s1, z2z2)
114     s1.Mod(s1, curve.P)
115     s2 := new(big.Int).Mul(y2, z1)
116     s2.Mul(s2, z1z1)
117     s2.Mod(s2, curve.P)
118     r := new(big.Int).Sub(s2, s1)
119     if r.Sign() == -1 {
120         r.Add(r, curve.P)
121     }
122     r.Lsh(r, 1)
123     v := new(big.Int).Mul(u1, i)
124
125     x3 := new(big.Int).Set(r)
126     x3.Mul(x3, x3)
127     x3.Sub(x3, j)
128     x3.Sub(x3, v)
129     x3.Sub(x3, v)
130     x3.Mod(x3, curve.P)
131
132     y3 := new(big.Int).Set(r)
133     v.Sub(v, x3)
134     y3.Mul(y3, v)
135     s1.Mul(s1, j)
136     s1.Lsh(s1, 1)
137     y3.Sub(y3, s1)
138     y3.Mod(y3, curve.P)
139
140     z3 := new(big.Int).Add(z1, z2)

```

```

141         z3.Mul(z3, z3)
142         z3.Sub(z3, z1z1)
143         if z3.Sign() == -1 {
144             z3.Add(z3, curve.P)
145         }
146         z3.Sub(z3, z2z2)
147         if z3.Sign() == -1 {
148             z3.Add(z3, curve.P)
149         }
150         z3.Mul(z3, h)
151         z3.Mod(z3, curve.P)
152
153         return x3, y3, z3
154     }
155
156     func (curve *CurveParams) Double(x1, y1 *big.Int) (*big.Int,
157         z1 := new(big.Int).SetInt64(1)
158         return curve.affineFromJacobian(curve.doubleJacobian
159     }
160
161     // doubleJacobian takes a point in Jacobian coordinates, (x,
162     // returns its double, also in Jacobian form.
163     func (curve *CurveParams) doubleJacobian(x, y, z *big.Int) (
164         // See http://hyperelliptic.org/EFD/g1p/auto-shortw-
165         delta := new(big.Int).Mul(z, z)
166         delta.Mod(delta, curve.P)
167         gamma := new(big.Int).Mul(y, y)
168         gamma.Mod(gamma, curve.P)
169         alpha := new(big.Int).Sub(x, delta)
170         if alpha.Sign() == -1 {
171             alpha.Add(alpha, curve.P)
172         }
173         alpha2 := new(big.Int).Add(x, delta)
174         alpha.Mul(alpha, alpha2)
175         alpha2.Set(alpha)
176         alpha.Lsh(alpha, 1)
177         alpha.Add(alpha, alpha2)
178
179         beta := alpha2.Mul(x, gamma)
180
181         x3 := new(big.Int).Mul(alpha, alpha)
182         beta8 := new(big.Int).Lsh(beta, 3)
183         x3.Sub(x3, beta8)
184         for x3.Sign() == -1 {
185             x3.Add(x3, curve.P)
186         }
187         x3.Mod(x3, curve.P)
188
189         z3 := new(big.Int).Add(y, z)

```

```

190         z3.Mul(z3, z3)
191         z3.Sub(z3, gamma)
192         if z3.Sign() == -1 {
193             z3.Add(z3, curve.P)
194         }
195         z3.Sub(z3, delta)
196         if z3.Sign() == -1 {
197             z3.Add(z3, curve.P)
198         }
199         z3.Mod(z3, curve.P)
200
201         beta.Lsh(beta, 2)
202         beta.Sub(beta, x3)
203         if beta.Sign() == -1 {
204             beta.Add(beta, curve.P)
205         }
206         y3 := alpha.Mul(alpha, beta)
207
208         gamma.Mul(gamma, gamma)
209         gamma.Lsh(gamma, 3)
210         gamma.Mod(gamma, curve.P)
211
212         y3.Sub(y3, gamma)
213         if y3.Sign() == -1 {
214             y3.Add(y3, curve.P)
215         }
216         y3.Mod(y3, curve.P)
217
218         return x3, y3, z3
219     }
220
221     func (curve *CurveParams) ScalarMult(Bx, By *big.Int, k []byte
222         // We have a slight problem in that the identity of
223         // point at infinity) cannot be represented in (x, y
224         // machine. Thus the standard add/double algorithm h
225         // slightly: our initial state is not the identity,
226         // ignore the first true bit in |k|. If we don't fi
227         // |k|, then we return nil, nil, because we cannot r
228         // element.
229
230         Bz := new(big.Int).SetInt64(1)
231         x := Bx
232         y := By
233         z := Bz
234
235         seenFirstTrue := false
236         for _, byte := range k {
237             for bitNum := 0; bitNum < 8; bitNum++ {
238                 if seenFirstTrue {
239                     x, y, z = curve.doubleJacobi

```

```

240         }
241         if byte&0x80 == 0x80 {
242             if !seenFirstTrue {
243                 seenFirstTrue = true
244             } else {
245                 x, y, z = curve.addJ
246             }
247         }
248         byte <<= 1
249     }
250 }
251
252 if !seenFirstTrue {
253     return nil, nil
254 }
255
256 return curve.affineFromJacobian(x, y, z)
257 }
258
259 func (curve *CurveParams) ScalarBaseMult(k []byte) (*big.Int
260     return curve.ScalarMult(curve.Gx, curve.Gy, k)
261 }
262
263 var mask = []byte{0xff, 0x1, 0x3, 0x7, 0xf, 0x1f, 0x3f, 0x7f
264
265 // GenerateKey returns a public/private key pair. The privat
266 // generated using the given reader, which must return rando
267 func GenerateKey(curve Curve, rand io.Reader) (priv []byte,
268     bitSize := curve.Params().BitSize
269     byteLen := (bitSize + 7) >> 3
270     priv = make([]byte, byteLen)
271
272     for x == nil {
273         _, err = io.ReadFull(rand, priv)
274         if err != nil {
275             return
276         }
277         // We have to mask off any excess bits in th
278         // underlying field is not a whole number of
279         priv[0] &= mask[bitSize%8]
280         // This is because, in tests, rand will retu
281         // want to get the point at infinity and loc
282         priv[1] ^= 0x42
283         x, y = curve.ScalarBaseMult(priv)
284     }
285     return
286 }
287
288 // Marshal converts a point into the form specified in secti

```

```

289 func Marshal(curve Curve, x, y *big.Int) []byte {
290     byteLen := (curve.Params().BitSize + 7) >> 3
291
292     ret := make([]byte, 1+2*byteLen)
293     ret[0] = 4 // uncompressed point
294
295     xBytes := x.Bytes()
296     copy(ret[1+byteLen-len(xBytes):], xBytes)
297     yBytes := y.Bytes()
298     copy(ret[1+2*byteLen-len(yBytes):], yBytes)
299     return ret
300 }
301
302 // Unmarshal converts a point, serialized by Marshal, into a
303 func Unmarshal(curve Curve, data []byte) (x, y *big.Int) {
304     byteLen := (curve.Params().BitSize + 7) >> 3
305     if len(data) != 1+2*byteLen {
306         return
307     }
308     if data[0] != 4 { // uncompressed form
309         return
310     }
311     x = new(big.Int).SetBytes(data[1 : 1+byteLen])
312     y = new(big.Int).SetBytes(data[1+byteLen:])
313     return
314 }
315
316 var initonce sync.Once
317 var p256 *CurveParams
318 var p384 *CurveParams
319 var p521 *CurveParams
320
321 func initAll() {
322     initP224()
323     initP256()
324     initP384()
325     initP521()
326 }
327
328 func initP256() {
329     // See FIPS 186-3, section D.2.3
330     p256 = new(CurveParams)
331     p256.P, _ = new(big.Int).SetString("1157920892103562
332     p256.N, _ = new(big.Int).SetString("1157920892103562
333     p256.B, _ = new(big.Int).SetString("5ac635d8aa3a93e7
334     p256.Gx, _ = new(big.Int).SetString("6b17d1f2e12c424
335     p256.Gy, _ = new(big.Int).SetString("4fe342e2fe1a7f9
336     p256.BitSize = 256
337 }

```

```

338
339 func initP384() {
340     // See FIPS 186-3, section D.2.4
341     p384 = new(CurveParams)
342     p384.P, _ = new(big.Int).SetString("3940200619639447
343     p384.N, _ = new(big.Int).SetString("3940200619639447
344     p384.B, _ = new(big.Int).SetString("b3312fa7e23ee7e4
345     p384.Gx, _ = new(big.Int).SetString("aa87ca22be8b053
346     p384.Gy, _ = new(big.Int).SetString("3617de4a96262c6
347     p384.BitSize = 384
348 }
349
350 func initP521() {
351     // See FIPS 186-3, section D.2.5
352     p521 = new(CurveParams)
353     p521.P, _ = new(big.Int).SetString("6864797660130609
354     p521.N, _ = new(big.Int).SetString("6864797660130609
355     p521.B, _ = new(big.Int).SetString("051953eb9618e1c9
356     p521.Gx, _ = new(big.Int).SetString("c6858e06b70404e
357     p521.Gy, _ = new(big.Int).SetString("11839296a789a3b
358     p521.BitSize = 521
359 }
360
361 // P256 returns a Curve which implements P-256 (see FIPS 186
362 func P256() Curve {
363     initonce.Do(initAll)
364     return p256
365 }
366
367 // P384 returns a Curve which implements P-384 (see FIPS 186
368 func P384() Curve {
369     initonce.Do(initAll)
370     return p384
371 }
372
373 // P521 returns a Curve which implements P-521 (see FIPS 186
374 func P521() Curve {
375     initonce.Do(initAll)
376     return p521
377 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/elliptic/p224.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package elliptic
6
7 // This is a constant-time, 32-bit implementation of P224. S
8 // section D.2.2.
9 //
10 // See http://www.imperialviolet.org/2010/12/04/ecc.html ([1
11
12 import (
13     "math/big"
14 )
15
16 var p224 p224Curve
17
18 type p224Curve struct {
19     *CurveParams
20     gx, gy, b p224FieldElement
21 }
22
23 func initP224() {
24     // See FIPS 186-3, section D.2.2
25     p224.CurveParams = new(CurveParams)
26     p224.P, _ = new(big.Int).SetString("2695994666715063
27     p224.N, _ = new(big.Int).SetString("2695994666715063
28     p224.B, _ = new(big.Int).SetString("b4050a850c04b3ab
29     p224.Gx, _ = new(big.Int).SetString("b70e0cbd6bb4bf7
30     p224.Gy, _ = new(big.Int).SetString("bd376388b5f723f
31     p224.BitSize = 224
32
33     p224.FromBig(&p224.gx, p224.Gx)
34     p224.FromBig(&p224.gy, p224.Gy)
35     p224.FromBig(&p224.b, p224.B)
36 }
37
38 // P224 returns a Curve which implements P-224 (see FIPS 186
39 func P224() Curve {
40     initonce.Do(initAll)
41     return p224
```

```

42 }
43
44 func (curve p224Curve) Params() *CurveParams {
45     return curve.CurveParams
46 }
47
48 func (curve p224Curve) IsOnCurve(bigX, bigY *big.Int) bool {
49     var x, y p224FieldElement
50     p224FromBig(&x, bigX)
51     p224FromBig(&y, bigY)
52
53     //  $y^2 = x^3 - 3x + b$ 
54     var tmp p224LargeFieldElement
55     var x3 p224FieldElement
56     p224Square(&x3, &x, &tmp)
57     p224Mul(&x3, &x3, &x, &tmp)
58
59     for i := 0; i < 8; i++ {
60         x[i] *= 3
61     }
62     p224Sub(&x3, &x3, &x)
63     p224Reduce(&x3)
64     p224Add(&x3, &x3, &curve.b)
65     p224Contract(&x3, &x3)
66
67     p224Square(&y, &y, &tmp)
68     p224Contract(&y, &y)
69
70     for i := 0; i < 8; i++ {
71         if y[i] != x3[i] {
72             return false
73         }
74     }
75     return true
76 }
77
78 func (p224Curve) Add(bigX1, bigY1, bigX2, bigY2 *big.Int) (x
79     var x1, y1, z1, x2, y2, z2, x3, y3, z3 p224FieldElem
80
81     p224FromBig(&x1, bigX1)
82     p224FromBig(&y1, bigY1)
83     z1[0] = 1
84     p224FromBig(&x2, bigX2)
85     p224FromBig(&y2, bigY2)
86     z2[0] = 1
87
88     p224AddJacobian(&x3, &y3, &z3, &x1, &y1, &z1, &x2, &
89     return p224ToAffine(&x3, &y3, &z3)
90 }
91

```

```

92 func (p224Curve) Double(bigX1, bigY1 *big.Int) (x, y *big.In
93     var x1, y1, z1, x2, y2, z2 p224FieldElement
94
95     p224FromBig(&x1, bigX1)
96     p224FromBig(&y1, bigY1)
97     z1[0] = 1
98
99     p224DoubleJacobian(&x2, &y2, &z2, &x1, &y1, &z1)
100    return p224ToAffine(&x2, &y2, &z2)
101 }
102
103 func (p224Curve) ScalarMult(bigX1, bigY1 *big.Int, scalar []
104     var x1, y1, z1, x2, y2, z2 p224FieldElement
105
106     p224FromBig(&x1, bigX1)
107     p224FromBig(&y1, bigY1)
108     z1[0] = 1
109
110     p224ScalarMult(&x2, &y2, &z2, &x1, &y1, &z1, scalar)
111    return p224ToAffine(&x2, &y2, &z2)
112 }
113
114 func (curve p224Curve) ScalarBaseMult(scalar []byte) (x, y *
115     var z1, x2, y2, z2 p224FieldElement
116
117     z1[0] = 1
118     p224ScalarMult(&x2, &y2, &z2, &curve.gx, &curve.gy,
119     return p224ToAffine(&x2, &y2, &z2)
120 }
121
122 // Field element functions.
123 //
124 // The field that we're dealing with is  $\mathbb{Z}/p\mathbb{Z}$  where  $p = 2^{22}$ 
125 //
126 // Field elements are represented by a FieldElement, which i
127 // array of 8 uint32's. The value of a FieldElement, a, is:
128 //  $a[0] + 2^{28} \cdot a[1] + 2^{56} \cdot a[2] + \dots + 2^{196} \cdot a[7]$ 
129 //
130 // Using 28-bit limbs means that there's only 4 bits of head
131 // than we would really like. But it has the useful feature
132 // exactly, making the reflections during a reduce much nice
133 type p224FieldElement [8]uint32
134
135 // p224Add computes *out = a+b
136 //
137 //  $a[i] + b[i] < 2^{32}$ 
138 func p224Add(out, a, b *p224FieldElement) {
139     for i := 0; i < 8; i++ {
140         out[i] = a[i] + b[i]

```

```

141     }
142 }
143
144 const two31p3 = 1<<31 + 1<<3
145 const two31m3 = 1<<31 - 1<<3
146 const two31m15m3 = 1<<31 - 1<<15 - 1<<3
147
148 // p224ZeroModP31 is 0 mod p where bit 31 is set in all limb
149 // subtract smaller amounts without underflow. See the secti
150 // [1] for reasoning.
151 var p224ZeroModP31 = []uint32{two31p3, two31m3, two31m3, two
152
153 // p224Sub computes *out = a-b
154 //
155 // a[i], b[i] < 2**30
156 // out[i] < 2**32
157 func p224Sub(out, a, b *p224FieldElement) {
158     for i := 0; i < 8; i++ {
159         out[i] = a[i] + p224ZeroModP31[i] - b[i]
160     }
161 }
162
163 // LargeFieldElement also represents an element of the field
164 // still spaced 28-bits apart and in little-endian order. So
165 // 0, 28, 56, ..., 392 bits, each 64-bits wide.
166 type p224LargeFieldElement [15]uint64
167
168 const two63p35 = 1<<63 + 1<<35
169 const two63m35 = 1<<63 - 1<<35
170 const two63m35m19 = 1<<63 - 1<<35 - 1<<19
171
172 // p224ZeroModP63 is 0 mod p where bit 63 is set in all limb
173 // "Subtraction" in [1] for why.
174 var p224ZeroModP63 = [8]uint64{two63p35, two63m35, two63m35,
175
176 const bottom12Bits = 0xfff
177 const bottom28Bits = 0xffffffff
178
179 // p224Mul computes *out = a*b
180 //
181 // a[i] < 2**29, b[i] < 2**30 (or vice versa)
182 // out[i] < 2**29
183 func p224Mul(out, a, b *p224FieldElement, tmp *p224LargeFiel
184     for i := 0; i < 15; i++ {
185         tmp[i] = 0
186     }
187
188     for i := 0; i < 8; i++ {
189         for j := 0; j < 8; j++ {

```

```

190             tmp[i+j] += uint64(a[i]) * uint64(b[
191         }
192     }
193
194     p224ReduceLarge(out, tmp)
195 }
196
197 // Square computes *out = a*a
198 //
199 // a[i] < 2**29
200 // out[i] < 2**29
201 func p224Square(out, a *p224FieldElement, tmp *p224LargeFiel
202     for i := 0; i < 15; i++ {
203         tmp[i] = 0
204     }
205
206     for i := 0; i < 8; i++ {
207         for j := 0; j <= i; j++ {
208             r := uint64(a[i]) * uint64(a[j])
209             if i == j {
210                 tmp[i+j] += r
211             } else {
212                 tmp[i+j] += r << 1
213             }
214         }
215     }
216
217     p224ReduceLarge(out, tmp)
218 }
219
220 // ReduceLarge converts a p224LargeFieldElement to a p224Fie
221 //
222 // in[i] < 2**62
223 func p224ReduceLarge(out *p224FieldElement, in *p224LargeFie
224     for i := 0; i < 8; i++ {
225         in[i] += p224ZeroModP63[i]
226     }
227
228     // Eliminate the coefficients at 2**224 and greater.
229     for i := 14; i >= 8; i-- {
230         in[i-8] -= in[i]
231         in[i-5] += (in[i] & 0xffff) << 12
232         in[i-4] += in[i] >> 16
233     }
234     in[8] = 0
235     // in[0..8] < 2**64
236
237     // As the values become small enough, we start to st
238     // and use 32-bit operations.
239     for i := 1; i < 8; i++ {

```

```

240             in[i+1] += in[i] >> 28
241             out[i] = uint32(in[i] & bottom28Bits)
242         }
243         in[0] -= in[8]
244         out[3] += uint32(in[8]&0xffff) << 12
245         out[4] += uint32(in[8] >> 16)
246         // in[0] < 2**64
247         // out[3] < 2**29
248         // out[4] < 2**29
249         // out[1,2,5..7] < 2**28
250
251         out[0] = uint32(in[0] & bottom28Bits)
252         out[1] += uint32((in[0] >> 28) & bottom28Bits)
253         out[2] += uint32(in[0] >> 56)
254         // out[0] < 2**28
255         // out[1..4] < 2**29
256         // out[5..7] < 2**28
257     }
258
259     // Reduce reduces the coefficients of a to smaller bounds.
260     //
261     // On entry: a[i] < 2**31 + 2**30
262     // On exit: a[i] < 2**29
263     func p224Reduce(a *p224FieldElement) {
264         for i := 0; i < 7; i++ {
265             a[i+1] += a[i] >> 28
266             a[i] &= bottom28Bits
267         }
268         top := a[7] >> 28
269         a[7] &= bottom28Bits
270
271         // top < 2**4
272         mask := top
273         mask |= mask >> 2
274         mask |= mask >> 1
275         mask <=<= 31
276         mask = uint32(int32(mask) >> 31)
277         // Mask is all ones if top != 0, all zero otherwise
278
279         a[0] -= top
280         a[3] += top << 12
281
282         // We may have just made a[0] negative but, if we di
283         // have added something to a[3], this it's > 2**12.
284         // carry down to a[0].
285         a[3] -= 1 & mask
286         a[2] += mask & (1<<28 - 1)
287         a[1] += mask & (1<<28 - 1)
288         a[0] += mask & (1 << 28)

```

```

289 }
290
291 // p224Invert calculates *out = in**(-1) by computing in**(2**
292 // i.e. Fermat's little theorem.
293 func p224Invert(out, in *p224FieldElement) {
294     var f1, f2, f3, f4 p224FieldElement
295     var c p224LargeFieldElement
296
297     p224Square(&f1, in, &c) // 2
298     p224Mul(&f1, &f1, in, &c) // 2**2 - 1
299     p224Square(&f1, &f1, &c) // 2**3 - 2
300     p224Mul(&f1, &f1, in, &c) // 2**3 - 1
301     p224Square(&f2, &f1, &c) // 2**4 - 2
302     p224Square(&f2, &f2, &c) // 2**5 - 4
303     p224Square(&f2, &f2, &c) // 2**6 - 8
304     p224Mul(&f1, &f1, &f2, &c) // 2**6 - 1
305     p224Square(&f2, &f1, &c) // 2**7 - 2
306     for i := 0; i < 5; i++ { // 2**12 - 2**6
307         p224Square(&f2, &f2, &c)
308     }
309     p224Mul(&f2, &f2, &f1, &c) // 2**12 - 1
310     p224Square(&f3, &f2, &c) // 2**13 - 2
311     for i := 0; i < 11; i++ { // 2**24 - 2**12
312         p224Square(&f3, &f3, &c)
313     }
314     p224Mul(&f2, &f3, &f2, &c) // 2**24 - 1
315     p224Square(&f3, &f2, &c) // 2**25 - 2
316     for i := 0; i < 23; i++ { // 2**48 - 2**24
317         p224Square(&f3, &f3, &c)
318     }
319     p224Mul(&f3, &f3, &f2, &c) // 2**48 - 1
320     p224Square(&f4, &f3, &c) // 2**49 - 2
321     for i := 0; i < 47; i++ { // 2**96 - 2**48
322         p224Square(&f4, &f4, &c)
323     }
324     p224Mul(&f3, &f3, &f4, &c) // 2**96 - 1
325     p224Square(&f4, &f3, &c) // 2**97 - 2
326     for i := 0; i < 23; i++ { // 2**120 - 2**24
327         p224Square(&f4, &f4, &c)
328     }
329     p224Mul(&f2, &f4, &f2, &c) // 2**120 - 1
330     for i := 0; i < 6; i++ { // 2**126 - 2**6
331         p224Square(&f2, &f2, &c)
332     }
333     p224Mul(&f1, &f1, &f2, &c) // 2**126 - 1
334     p224Square(&f1, &f1, &c) // 2**127 - 2
335     p224Mul(&f1, &f1, in, &c) // 2**127 - 1
336     for i := 0; i < 97; i++ { // 2**224 - 2**97
337         p224Square(&f1, &f1, &c)

```

```

338     }
339     p224Mul(out, &f1, &f3, &c) // 2**224 - 2**96 - 1
340 }
341
342 // p224Contract converts a FieldElement to its unique, minim
343 //
344 // On entry, in[i] < 2**29
345 // On exit, in[i] < 2**28
346 func p224Contract(out, in *p224FieldElement) {
347     copy(out[:], in[:])
348
349     for i := 0; i < 7; i++ {
350         out[i+1] += out[i] >> 28
351         out[i] &= bottom28Bits
352     }
353     top := out[7] >> 28
354     out[7] &= bottom28Bits
355
356     out[0] -= top
357     out[3] += top << 12
358
359     // We may just have made out[i] negative. So we carr
360     // out[0] negative then we know that out[3] is suffi
361     // because we just added to it.
362     for i := 0; i < 3; i++ {
363         mask := uint32(int32(out[i]) >> 31)
364         out[i] += (1 << 28) & mask
365         out[i+1] -= 1 & mask
366     }
367
368     // We might have pushed out[3] over 2**28 so we perf
369     // carry chain.
370     for i := 3; i < 7; i++ {
371         out[i+1] += out[i] >> 28
372         out[i] &= bottom28Bits
373     }
374     top = out[7] >> 28
375     out[7] &= bottom28Bits
376
377     // Eliminate top while maintaining the same value mo
378     out[0] -= top
379     out[3] += top << 12
380
381     // There are two cases to consider for out[3]:
382     // 1) The first time that we eliminated top, we di
383     // 2**28. In this case, the partial carry chain
384     // and top is zero.
385     // 2) We did push out[3] over 2**28 the first time
386     // The first value of top was in [0..16), there
387     // the first top, 0xfff1000 <= out[3] <= 0xffff

```

```

388 //      overflowing and being reduced by the second
389 //      0xf000. Thus it cannot have overflowed when
390 //      second time.
391
392 // Again, we may just have made out[0] negative, so
393 // As before, if we made out[0] negative then we know
394 // sufficiently positive.
395 for i := 0; i < 3; i++ {
396     mask := uint32(int32(out[i]) >> 31)
397     out[i] += (1 << 28) & mask
398     out[i+1] -= 1 & mask
399 }
400
401 // Now we see if the value is >= p and, if so, subtract
402
403 // First we build a mask from the top four limbs, which
404 // equal to bottom28Bits if the whole value is >= p.
405 // ends up with any zero bits in the bottom 28 bits,
406 // true.
407 top4AllOnes := uint32(0xffffffff)
408 for i := 4; i < 8; i++ {
409     top4AllOnes &= (out[i] & bottom28Bits) - 1
410 }
411 top4AllOnes |= 0xf0000000
412 // Now we replicate any zero bits to all the bits in
413 top4AllOnes &= top4AllOnes >> 16
414 top4AllOnes &= top4AllOnes >> 8
415 top4AllOnes &= top4AllOnes >> 4
416 top4AllOnes &= top4AllOnes >> 2
417 top4AllOnes &= top4AllOnes >> 1
418 top4AllOnes = uint32(int32(top4AllOnes<<31) >> 31)
419
420 // Now we test whether the bottom three limbs are non-zero
421 bottom3NonZero := out[0] | out[1] | out[2]
422 bottom3NonZero |= bottom3NonZero >> 16
423 bottom3NonZero |= bottom3NonZero >> 8
424 bottom3NonZero |= bottom3NonZero >> 4
425 bottom3NonZero |= bottom3NonZero >> 2
426 bottom3NonZero |= bottom3NonZero >> 1
427 bottom3NonZero = uint32(int32(bottom3NonZero<<31) >> 31)
428
429 // Everything depends on the value of out[3].
430 // If it's > 0xffff000 and top4AllOnes != 0 then
431 // If it's = 0xffff000 and top4AllOnes != 0 and bottom3NonZero
432 // then the whole value is >= p
433 // If it's < 0xffff000, then the whole value is < p
434 n := out[3] - 0xffff000
435 out3Equal := n
436 out3Equal |= out3Equal >> 16

```

```

437         out3Equal |= out3Equal >> 8
438         out3Equal |= out3Equal >> 4
439         out3Equal |= out3Equal >> 2
440         out3Equal |= out3Equal >> 1
441         out3Equal = ^uint32(int32(out3Equal<<31) >> 31)
442
443         // If out[3] > 0xffff000 then n's MSB will be zero.
444         out3GT := ^uint32(int32(n<<31) >> 31)
445
446         mask := top4AllOnes & ((out3Equal & bottom3NonZero)
447         out[0] -= 1 & mask
448         out[3] -= 0xffff000 & mask
449         out[4] -= 0xffffffff & mask
450         out[5] -= 0xffffffff & mask
451         out[6] -= 0xffffffff & mask
452         out[7] -= 0xffffffff & mask
453     }
454
455     // Group element functions.
456     //
457     // These functions deal with group elements. The group is an
458     // group with a = -3 defined in FIPS 186-3, section D.2.2.
459
460     // p224AddJacobian computes *out = a+b where a != b.
461     func p224AddJacobian(x3, y3, z3, x1, y1, z1, x2, y2, z2 *p22
462         // See http://hyperelliptic.org/EFD/g1p/auto-shortw-
463         var z1z1, z2z2, u1, u2, s1, s2, h, i, j, r, v p224Fi
464         var c p224LargeFieldElement
465
466         // Z1Z1 = Z12
467         p224Square(&z1z1, z1, &c)
468         // Z2Z2 = Z22
469         p224Square(&z2z2, z2, &c)
470         // U1 = X1*Z2Z2
471         p224Mul(&u1, x1, &z2z2, &c)
472         // U2 = X2*Z1Z1
473         p224Mul(&u2, x2, &z1z1, &c)
474         // S1 = Y1*Z2*Z2Z2
475         p224Mul(&s1, z2, &z2z2, &c)
476         p224Mul(&s1, y1, &s1, &c)
477         // S2 = Y2*Z1*Z1Z1
478         p224Mul(&s2, z1, &z1z1, &c)
479         p224Mul(&s2, y2, &s2, &c)
480         // H = U2-U1
481         p224Sub(&h, &u2, &u1)
482         p224Reduce(&h)
483         // I = (2*H)2
484         for j := 0; j < 8; j++ {
485             i[j] = h[j] << 1

```

```

486     }
487     p224Reduce(&i)
488     p224Square(&i, &i, &c)
489     // J = H*I
490     p224Mul(&j, &h, &i, &c)
491     // r = 2*(S2-S1)
492     p224Sub(&r, &s2, &s1)
493     p224Reduce(&r)
494     for i := 0; i < 8; i++ {
495         r[i] <= 1
496     }
497     p224Reduce(&r)
498     // V = U1*I
499     p224Mul(&v, &u1, &i, &c)
500     // Z3 = ((Z1+Z2)^2-Z1Z1-Z2Z2)*H
501     p224Add(&z1z1, &z1z1, &z2z2)
502     p224Add(&z2z2, z1, z2)
503     p224Reduce(&z2z2)
504     p224Square(&z2z2, &z2z2, &c)
505     p224Sub(z3, &z2z2, &z1z1)
506     p224Reduce(z3)
507     p224Mul(z3, z3, &h, &c)
508     // X3 = r^2-J-2*V
509     for i := 0; i < 8; i++ {
510         z1z1[i] = v[i] << 1
511     }
512     p224Add(&z1z1, &j, &z1z1)
513     p224Reduce(&z1z1)
514     p224Square(x3, &r, &c)
515     p224Sub(x3, x3, &z1z1)
516     p224Reduce(x3)
517     // Y3 = r*(V-X3)-2*S1*J
518     for i := 0; i < 8; i++ {
519         s1[i] <= 1
520     }
521     p224Mul(&s1, &s1, &j, &c)
522     p224Sub(&z1z1, &v, x3)
523     p224Reduce(&z1z1)
524     p224Mul(&z1z1, &z1z1, &r, &c)
525     p224Sub(y3, &z1z1, &s1)
526     p224Reduce(y3)
527 }
528
529 // p224DoubleJacobian computes *out = a+a.
530 func p224DoubleJacobian(x3, y3, z3, x1, y1, z1 *p224FieldEle
531     var delta, gamma, beta, alpha, t p224FieldElement
532     var c p224LargeFieldElement
533
534     p224Square(&delta, z1, &c)
535     p224Square(&gamma, y1, &c)

```

```

536     p224Mul(&beta, x1, &gamma, &c)
537
538     // alpha = 3*(X1-delta)*(X1+delta)
539     p224Add(&t, x1, &delta)
540     for i := 0; i < 8; i++ {
541         t[i] += t[i] << 1
542     }
543     p224Reduce(&t)
544     p224Sub(&alpha, x1, &delta)
545     p224Reduce(&alpha)
546     p224Mul(&alpha, &alpha, &t, &c)
547
548     // Z3 = (Y1+Z1)^2-gamma-delta
549     p224Add(z3, y1, z1)
550     p224Reduce(z3)
551     p224Square(z3, z3, &c)
552     p224Sub(z3, z3, &gamma)
553     p224Reduce(z3)
554     p224Sub(z3, z3, &delta)
555     p224Reduce(z3)
556
557     // X3 = alpha^2-8*beta
558     for i := 0; i < 8; i++ {
559         delta[i] = beta[i] << 3
560     }
561     p224Reduce(&delta)
562     p224Square(x3, &alpha, &c)
563     p224Sub(x3, x3, &delta)
564     p224Reduce(x3)
565
566     // Y3 = alpha*(4*beta-X3)-8*gamma^2
567     for i := 0; i < 8; i++ {
568         beta[i] <<= 2
569     }
570     p224Sub(&beta, &beta, x3)
571     p224Reduce(&beta)
572     p224Square(&gamma, &gamma, &c)
573     for i := 0; i < 8; i++ {
574         gamma[i] <<= 3
575     }
576     p224Reduce(&gamma)
577     p224Mul(y3, &alpha, &beta, &c)
578     p224Sub(y3, y3, &gamma)
579     p224Reduce(y3)
580 }
581
582 // p224CopyConditional sets *out = *in iff the least-signifi
583 // is true, and it runs in constant time.
584 func p224CopyConditional(out, in *p224FieldElement, control

```

```

585         control <= 31
586         control = uint32(int32(control) >> 31)
587
588         for i := 0; i < 8; i++ {
589             out[i] ^= (out[i] ^ in[i]) & control
590         }
591     }
592
593     func p224ScalarMult(outX, outY, outZ, inX, inY, inZ *p224Fie
594         var xx, yy, zz p224FieldElement
595         for i := 0; i < 8; i++ {
596             outZ[i] = 0
597         }
598
599         firstBit := uint32(1)
600         for _, byte := range scalar {
601             for bitNum := uint(0); bitNum < 8; bitNum++
602                 p224DoubleJacobian(outX, outY, outZ,
603                     bit := uint32((byte >> (7 - bitNum))
604                     p224AddJacobian(&xx, &yy, &zz, inX,
605                     p224CopyConditional(outX, inX, first
606                     p224CopyConditional(outY, inY, first
607                     p224CopyConditional(outZ, inZ, first
608                     p224CopyConditional(outX, &xx, ^firs
609                     p224CopyConditional(outY, &yy, ^firs
610                     p224CopyConditional(outZ, &zz, ^firs
611                     firstBit = firstBit & ^bit
612             }
613         }
614     }
615
616     // p224ToAffine converts from Jacobian to affine form.
617     func p224ToAffine(x, y, z *p224FieldElement) (*big.Int, *big
618         var zinv, zinvSq, outX, outY p224FieldElement
619         var tmp p224LargeFieldElement
620
621         isPointAtInfinity := true
622         for i := 0; i < 8; i++ {
623             if z[i] != 0 {
624                 isPointAtInfinity = false
625                 break
626             }
627         }
628
629         if isPointAtInfinity {
630             return nil, nil
631         }
632
633         p224Invert(&zinv, z)

```

```

634     p224Square(&zinvSq, &zinv, &tmp)
635     p224Mul(x, x, &zinvSq, &tmp)
636     p224Mul(&zinvSq, &zinvSq, &zinv, &tmp)
637     p224Mul(y, y, &zinvSq, &tmp)
638
639     p224Contract(&outx, x)
640     p224Contract(&outy, y)
641     return p224ToBig(&outx), p224ToBig(&outy)
642 }
643
644 // get28BitsFromEnd returns the least-significant 28 bits fr
645 // where buf is interpreted as a big-endian number.
646 func get28BitsFromEnd(buf []byte, shift uint) (uint32, []byte)
647     var ret uint32
648
649     for i := uint(0); i < 4; i++ {
650         var b byte
651         if l := len(buf); l > 0 {
652             b = buf[l-1]
653             // We don't remove the byte if we're
654             // reading all of it.
655             if i != 3 || shift == 4 {
656                 buf = buf[:l-1]
657             }
658         }
659         ret |= uint32(b) << (8 * i) >> shift
660     }
661     ret &= bottom28Bits
662     return ret, buf
663 }
664
665 // p224FromBig sets *out = *in.
666 func p224FromBig(out *p224FieldElement, in *big.Int) {
667     bytes := in.Bytes()
668     out[0], bytes = get28BitsFromEnd(bytes, 0)
669     out[1], bytes = get28BitsFromEnd(bytes, 4)
670     out[2], bytes = get28BitsFromEnd(bytes, 0)
671     out[3], bytes = get28BitsFromEnd(bytes, 4)
672     out[4], bytes = get28BitsFromEnd(bytes, 0)
673     out[5], bytes = get28BitsFromEnd(bytes, 4)
674     out[6], bytes = get28BitsFromEnd(bytes, 0)
675     out[7], bytes = get28BitsFromEnd(bytes, 4)
676 }
677
678 // p224ToBig returns in as a big.Int.
679 func p224ToBig(in *p224FieldElement) *big.Int {
680     var buf [28]byte
681     buf[27] = byte(in[0])
682     buf[26] = byte(in[0] >> 8)
683     buf[25] = byte(in[0] >> 16)

```

```

684         buf[24] = byte(((in[0] >> 24) & 0x0f) | (in[1]<<4)&0
685
686         buf[23] = byte(in[1] >> 4)
687         buf[22] = byte(in[1] >> 12)
688         buf[21] = byte(in[1] >> 20)
689
690         buf[20] = byte(in[2])
691         buf[19] = byte(in[2] >> 8)
692         buf[18] = byte(in[2] >> 16)
693         buf[17] = byte(((in[2] >> 24) & 0x0f) | (in[3]<<4)&0
694
695         buf[16] = byte(in[3] >> 4)
696         buf[15] = byte(in[3] >> 12)
697         buf[14] = byte(in[3] >> 20)
698
699         buf[13] = byte(in[4])
700         buf[12] = byte(in[4] >> 8)
701         buf[11] = byte(in[4] >> 16)
702         buf[10] = byte(((in[4] >> 24) & 0x0f) | (in[5]<<4)&0
703
704         buf[9] = byte(in[5] >> 4)
705         buf[8] = byte(in[5] >> 12)
706         buf[7] = byte(in[5] >> 20)
707
708         buf[6] = byte(in[6])
709         buf[5] = byte(in[6] >> 8)
710         buf[4] = byte(in[6] >> 16)
711         buf[3] = byte(((in[6] >> 24) & 0x0f) | (in[7]<<4)&0x
712
713         buf[2] = byte(in[7] >> 4)
714         buf[1] = byte(in[7] >> 12)
715         buf[0] = byte(in[7] >> 20)
716
717         return new(big.Int).SetBytes(buf[:])
718     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/hmac/hmac.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package hmac implements the Keyed-Hash Message Authentica
6 // defined in U.S. Federal Information Processing Standards
7 // An HMAC is a cryptographic hash that uses a key to sign a
8 // The receiver verifies the hash by recomputing it using th
9 package hmac
10
11 import (
12     "hash"
13 )
14
15 // FIPS 198:
16 // http://csrc.nist.gov/publications/fips/fips198/fips-198a.
17
18 // key is zero padded to the block size of the hash function
19 // ipad = 0x36 byte repeated for key length
20 // opad = 0x5c byte repeated for key length
21 // hmac = H([key ^ opad] H([key ^ ipad] text))
22
23 type hmac struct {
24     size          int
25     blocksize    int
26     key, tmp      []byte
27     outer, inner hash.Hash
28 }
29
30 func (h *hmac) tmpPad(xor byte) {
31     for i, k := range h.key {
32         h.tmp[i] = xor ^ k
33     }
34     for i := len(h.key); i < h.blocksize; i++ {
35         h.tmp[i] = xor
36     }
37 }
38
39 func (h *hmac) Sum(in []byte) []byte {
40     origLen := len(in)
41     in = h.inner.Sum(in)
```

```

42         h.tmpPad(0x5c)
43         copy(h.tmp[h.blocksize:], in[origLen:])
44         h.outer.Reset()
45         h.outer.Write(h.tmp)
46         return h.outer.Sum(in[:origLen])
47     }
48
49     func (h *hmac) Write(p []byte) (n int, err error) {
50         return h.inner.Write(p)
51     }
52
53     func (h *hmac) Size() int { return h.size }
54
55     func (h *hmac) BlockSize() int { return h.blocksize }
56
57     func (h *hmac) Reset() {
58         h.inner.Reset()
59         h.tmpPad(0x36)
60         h.inner.Write(h.tmp[0:h.blocksize])
61     }
62
63     // New returns a new HMAC hash using the given hash.Hash typ
64     func New(h func() hash.Hash, key []byte) hash.Hash {
65         hm := new(hmac)
66         hm.outer = h()
67         hm.inner = h()
68         hm.size = hm.inner.Size()
69         hm.blocksize = hm.inner.BlockSize()
70         hm.tmp = make([]byte, hm.blocksize+hm.size)
71         if len(key) > hm.blocksize {
72             // If key is too big, hash it.
73             hm.outer.Write(key)
74             key = hm.outer.Sum(nil)
75         }
76         hm.key = make([]byte, len(key))
77         copy(hm.key, key)
78         hm.Reset()
79         return hm
80     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/md5/md5.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package md5 implements the MD5 hash algorithm as defined
6 package md5
7
8 import (
9     "crypto"
10    "hash"
11 )
12
13 func init() {
14     crypto.RegisterHash(crypto.MD5, New)
15 }
16
17 // The size of an MD5 checksum in bytes.
18 const Size = 16
19
20 // The blocksize of MD5 in bytes.
21 const BlockSize = 64
22
23 const (
24     _Chunk = 64
25     _Init0 = 0x67452301
26     _Init1 = 0xEFCDAB89
27     _Init2 = 0x98BADCFE
28     _Init3 = 0x10325476
29 )
30
31 // digest represents the partial evaluation of a checksum.
32 type digest struct {
33     s    [4]uint32
34     x    [_Chunk]byte
35     nx   int
36     len  uint64
37 }
38
39 func (d *digest) Reset() {
40     d.s[0] = _Init0
41     d.s[1] = _Init1
```

```

42         d.s[2] = _Init2
43         d.s[3] = _Init3
44         d.nx = 0
45         d.len = 0
46     }
47
48     // New returns a new hash.Hash computing the MD5 checksum.
49     func New() hash.Hash {
50         d := new(digest)
51         d.Reset()
52         return d
53     }
54
55     func (d *digest) Size() int { return Size }
56
57     func (d *digest) BlockSize() int { return BlockSize }
58
59     func (d *digest) Write(p []byte) (nn int, err error) {
60         nn = len(p)
61         d.len += uint64(nn)
62         if d.nx > 0 {
63             n := len(p)
64             if n > _Chunk-d.nx {
65                 n = _Chunk - d.nx
66             }
67             for i := 0; i < n; i++ {
68                 d.x[d.nx+i] = p[i]
69             }
70             d.nx += n
71             if d.nx == _Chunk {
72                 _Block(d, d.x[0:])
73                 d.nx = 0
74             }
75             p = p[n:]
76         }
77         n := _Block(d, p)
78         p = p[n:]
79         if len(p) > 0 {
80             d.nx = copy(d.x[:], p)
81         }
82         return
83     }
84
85     func (d0 *digest) Sum(in []byte) []byte {
86         // Make a copy of d0 so that caller can keep writing
87         d := *d0
88
89         // Padding. Add a 1 bit and 0 bits until 56 bytes n
90         len := d.len
91         var tmp [64]byte

```

```

92         tmp[0] = 0x80
93         if len%64 < 56 {
94             d.Write(tmp[0 : 56-len%64])
95         } else {
96             d.Write(tmp[0 : 64+56-len%64])
97         }
98
99         // Length in bits.
100        len <= 3
101        for i := uint(0); i < 8; i++ {
102            tmp[i] = byte(len >> (8 * i))
103        }
104        d.Write(tmp[0:8])
105
106        if d.nx != 0 {
107            panic("d.nx != 0")
108        }
109
110        var digest [Size]byte
111        for i, s := range d.s {
112            digest[i*4] = byte(s)
113            digest[i*4+1] = byte(s >> 8)
114            digest[i*4+2] = byte(s >> 16)
115            digest[i*4+3] = byte(s >> 24)
116        }
117
118        return append(in, digest[:]...)
119    }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/md5/md5block.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // MD5 block step.
6 // In its own file so that a faster assembly or C version
7 // can be substituted easily.
8
9 package md5
10
11 // table[i] = int((1<<32) * abs(sin(i+1 radians))).
12 var table = []uint32{
13     // round 1
14     0xd76aa478,
15     0xe8c7b756,
16     0x242070db,
17     0xc1bdcee,
18     0xf57c0faf,
19     0x4787c62a,
20     0xa8304613,
21     0xfd469501,
22     0x698098d8,
23     0x8b44f7af,
24     0xffff5bb1,
25     0x895cd7be,
26     0x6b901122,
27     0xfd987193,
28     0xa679438e,
29     0x49b40821,
30
31     // round 2
32     0xf61e2562,
33     0xc040b340,
34     0x265e5a51,
35     0xe9b6c7aa,
36     0xd62f105d,
37     0x2441453,
38     0xd8a1e681,
39     0xe7d3fbc8,
40     0x21e1cde6,
41     0xc33707d6,
```

```
42         0xf4d50d87,
43         0x455a14ed,
44         0xa9e3e905,
45         0xfcefa3f8,
46         0x676f02d9,
47         0x8d2a4c8a,
48
49         // round3
50         0xfffa3942,
51         0x8771f681,
52         0x6d9d6122,
53         0xfde5380c,
54         0xa4beea44,
55         0x4bdecfa9,
56         0xf6bb4b60,
57         0xbefbfc70,
58         0x289b7ec6,
59         0xeea127fa,
60         0xd4ef3085,
61         0x4881d05,
62         0xd9d4d039,
63         0xe6db99e5,
64         0x1fa27cf8,
65         0xc4ac5665,
66
67         // round 4
68         0xf4292244,
69         0x432aff97,
70         0xab9423a7,
71         0xfc93a039,
72         0x655b59c3,
73         0x8f0ccc92,
74         0xffeff47d,
75         0x85845dd1,
76         0x6fa87e4f,
77         0xfe2ce6e0,
78         0xa3014314,
79         0x4e0811a1,
80         0xf7537e82,
81         0xbd3af235,
82         0x2ad7d2bb,
83         0xeb86d391,
84     }
85
86     var shift1 = []uint{7, 12, 17, 22}
87     var shift2 = []uint{5, 9, 14, 20}
88     var shift3 = []uint{4, 11, 16, 23}
89     var shift4 = []uint{6, 10, 15, 21}
90
91     func _Block(dig *digest, p []byte) int {
```

```

92     a := dig.s[0]
93     b := dig.s[1]
94     c := dig.s[2]
95     d := dig.s[3]
96     n := 0
97     var X [16]uint32
98     for len(p) >= _Chunk {
99         aa, bb, cc, dd := a, b, c, d
100
101         j := 0
102         for i := 0; i < 16; i++ {
103             X[i] = uint32(p[j]) | uint32(p[j+1])
104             j += 4
105         }
106
107         // If this needs to be made faster in the fu
108         // the usual trick is to unroll each of the
109         // loops by a factor of 4; that lets you rep
110         // the shift[] lookups with constants and,
111         // with suitable variable renaming in each
112         // unrolled body, delete the a, b, c, d = d,
113         // (or you can let the optimizer do the rena
114         //
115         // The index variables are uint so that % by
116         // of two can be optimized easily by a compi
117
118         // Round 1.
119         for i := uint(0); i < 16; i++ {
120             x := i
121             s := shift1[i%4]
122             f := ((c ^ d) & b) ^ d
123             a += f + X[x] + table[i]
124             a = a<<s | a>>(32-s) + b
125             a, b, c, d = d, a, b, c
126         }
127
128         // Round 2.
129         for i := uint(0); i < 16; i++ {
130             x := (1 + 5*i) % 16
131             s := shift2[i%4]
132             g := ((b ^ c) & d) ^ c
133             a += g + X[x] + table[i+16]
134             a = a<<s | a>>(32-s) + b
135             a, b, c, d = d, a, b, c
136         }
137
138         // Round 3.
139         for i := uint(0); i < 16; i++ {
140             x := (5 + 3*i) % 16

```

```

141             s := shift3[i%4]
142             h := b ^ c ^ d
143             a += h + X[x] + table[i+32]
144             a = a<<s | a>>(32-s) + b
145             a, b, c, d = d, a, b, c
146         }
147
148         // Round 4.
149         for i := uint(0); i < 16; i++ {
150             x := (7 * i) % 16
151             s := shift4[i%4]
152             j := c ^ (b | ^d)
153             a += j + X[x] + table[i+48]
154             a = a<<s | a>>(32-s) + b
155             a, b, c, d = d, a, b, c
156         }
157
158         a += aa
159         b += bb
160         c += cc
161         d += dd
162
163         p = p[_Chunk:]
164         n += _Chunk
165     }
166
167     dig.s[0] = a
168     dig.s[1] = b
169     dig.s[2] = c
170     dig.s[3] = d
171     return n
172 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/rand/rand.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package rand implements a cryptographically secure
6 // pseudorandom number generator.
7 package rand
8
9 import "io"
10
11 // Reader is a global, shared instance of a cryptographically
12 // strong pseudo-random generator.
13 // On Unix-like systems, Reader reads from /dev/urandom.
14 // On Windows systems, Reader uses the CryptGenRandom API.
15 var Reader io.Reader
16
17 // Read is a helper function that calls Reader.Read.
18 func Read(b []byte) (n int, err error) { return Reader.Read(b)}
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/rand/rand_unix.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 // Unix cryptographically secure pseudorandom number
8 // generator.
9
10 package rand
11
12 import (
13     "bufio"
14     "crypto/aes"
15     "crypto/cipher"
16     "io"
17     "os"
18     "sync"
19     "time"
20 )
21
22 // Easy implementation: read from /dev/urandom.
23 // This is sufficient on Linux, OS X, and FreeBSD.
24
25 func init() { Reader = &devReader{name: "/dev/urandom"} }
26
27 // A devReader satisfies reads by reading the file named name
28 type devReader struct {
29     name string
30     f     io.Reader
31     mu    sync.Mutex
32 }
33
34 func (r *devReader) Read(b []byte) (n int, err error) {
35     r.mu.Lock()
36     defer r.mu.Unlock()
37     if r.f == nil {
38         f, err := os.Open(r.name)
39         if f == nil {
40             return 0, err
41         }
42     }
```

```

42             r.f = bufio.NewReader(f)
43         }
44         return r.f.Read(b)
45     }
46
47     // Alternate pseudo-random implementation for use on
48     // systems without a reliable /dev/urandom. So far we
49     // haven't needed it.
50
51     // newReader returns a new pseudorandom generator that
52     // seeds itself by reading from entropy. If entropy == nil,
53     // the generator seeds itself by reading from the system's
54     // random number generator, typically /dev/random.
55     // The Read method on the returned reader always returns
56     // the full amount asked for, or else it returns an error.
57     //
58     // The generator uses the X9.31 algorithm with AES-128,
59     // reseeding after every 1 MB of generated data.
60     func newReader(entropy io.Reader) io.Reader {
61         if entropy == nil {
62             entropy = &devReader{name: "/dev/random"}
63         }
64         return &reader{entropy: entropy}
65     }
66
67     type reader struct {
68         mu                sync.Mutex
69         budget            int // number of bytes that can
70         cipher            cipher.Block
71         entropy            io.Reader
72         time, seed, dst, key [aes.BlockSize]byte
73     }
74
75     func (r *reader) Read(b []byte) (n int, err error) {
76         r.mu.Lock()
77         defer r.mu.Unlock()
78         n = len(b)
79
80         for len(b) > 0 {
81             if r.budget == 0 {
82                 _, err := io.ReadFull(r.entropy, r.s
83                 if err != nil {
84                     return n - len(b), err
85                 }
86                 _, err = io.ReadFull(r.entropy, r.ke
87                 if err != nil {
88                     return n - len(b), err
89                 }
90                 r.cipher, err = aes.NewCipher(r.key[
91                 if err != nil {

```

```

92             return n - len(b), err
93         }
94         r.budget = 1 << 20 // reseed after g
95     }
96     r.budget -= aes.BlockSize
97
98     // ANSI X9.31 (== X9.17) algorithm, but usin
99     //
100    // single block:
101    // t = encrypt(time)
102    // dst = encrypt(t^seed)
103    // seed = encrypt(t^dst)
104    ns := time.Now().UnixNano()
105    r.time[0] = byte(ns >> 56)
106    r.time[1] = byte(ns >> 48)
107    r.time[2] = byte(ns >> 40)
108    r.time[3] = byte(ns >> 32)
109    r.time[4] = byte(ns >> 24)
110    r.time[5] = byte(ns >> 16)
111    r.time[6] = byte(ns >> 8)
112    r.time[7] = byte(ns)
113    r.cipher.Encrypt(r.time[0:], r.time[0:])
114    for i := 0; i < aes.BlockSize; i++ {
115        r.dst[i] = r.time[i] ^ r.seed[i]
116    }
117    r.cipher.Encrypt(r.dst[0:], r.dst[0:])
118    for i := 0; i < aes.BlockSize; i++ {
119        r.seed[i] = r.time[i] ^ r.dst[i]
120    }
121    r.cipher.Encrypt(r.seed[0:], r.seed[0:])
122
123    m := copy(b, r.dst[0:])
124    b = b[m:]
125 }
126
127     return n, nil
128 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/rand/util.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package rand
6
7 import (
8     "errors"
9     "io"
10    "math/big"
11 )
12
13 // Prime returns a number, p, of the given size, such that p
14 // with high probability.
15 func Prime(rand io.Reader, bits int) (p *big.Int, err error) {
16     if bits < 1 {
17         err = errors.New("crypto/rand: prime size mu
18     }
19
20     b := uint(bits % 8)
21     if b == 0 {
22         b = 8
23     }
24
25     bytes := make([]byte, (bits+7)/8)
26     p = new(big.Int)
27
28     for {
29         _, err = io.ReadFull(rand, bytes)
30         if err != nil {
31             return nil, err
32         }
33
34         // Clear bits in the first byte to make sure
35         bytes[0] &= uint8(int(1<<b) - 1)
36         // Don't let the value be too small, i.e, se
37         bytes[0] |= 1 << (b - 1)
38         // Make the value odd since an even number t
39         bytes[len(bytes)-1] |= 1
40
41         p.SetBytes(bytes)
```

```

42             if p.ProbablyPrime(20) {
43                 return
44             }
45         }
46     }
47     return
48 }
49
50 // Int returns a uniform random value in [0, max).
51 func Int(rand io.Reader, max *big.Int) (n *big.Int, err error)
52     k := (max.BitLen() + 7) / 8
53
54     // b is the number of bits in the most significant b
55     b := uint(max.BitLen() % 8)
56     if b == 0 {
57         b = 8
58     }
59
60     bytes := make([]byte, k)
61     n = new(big.Int)
62
63     for {
64         _, err = io.ReadFull(rand, bytes)
65         if err != nil {
66             return nil, err
67         }
68
69         // Clear bits in the first byte to increase
70         // that the candidate is < max.
71         bytes[0] &= uint8(int(1<<b) - 1)
72
73         n.SetBytes(bytes)
74         if n.Cmp(max) < 0 {
75             return
76         }
77     }
78
79     return
80 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/crypto/rc4/rc4.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package rc4 implements RC4 encryption, as defined in Bruce
6 // Schneier's Applied Cryptography.
7 package rc4
8
9 // BUG(agl): RC4 is in common use but has design weaknesses
10 // it a poor choice for new protocols.
11
12 import "strconv"
13
14 // A Cipher is an instance of RC4 using a particular key.
15 type Cipher struct {
16     s    [256]byte
17     i, j uint8
18 }
19
20 type KeySizeError int
21
22 func (k KeySizeError) Error() string {
23     return "crypto/rc4: invalid key size " + strconv.Itoa
24 }
25
26 // NewCipher creates and returns a new Cipher. The key argu
27 // RC4 key, at least 1 byte and at most 256 bytes.
28 func NewCipher(key []byte) (*Cipher, error) {
29     k := len(key)
30     if k < 1 || k > 256 {
31         return nil, KeySizeError(k)
32     }
33     var c Cipher
34     for i := 0; i < 256; i++ {
35         c.s[i] = uint8(i)
36     }
37     var j uint8 = 0
38     for i := 0; i < 256; i++ {
39         j += c.s[i] + key[i%k]
40         c.s[i], c.s[j] = c.s[j], c.s[i]
41     }
42     return &c, nil
43 }
44
```

```

45 // XORKeyStream sets dst to the result of XORing src with th
46 // Dst and src may be the same slice but otherwise should no
47 func (c *Cipher) XORKeyStream(dst, src []byte) {
48     for i := range src {
49         c.i += 1
50         c.j += c.s[c.i]
51         c.s[c.i], c.s[c.j] = c.s[c.j], c.s[c.i]
52         dst[i] = src[i] ^ c.s[c.s[c.i]+c.s[c.j]]
53     }
54 }
55
56 // Reset zeros the key data so that it will no longer appear
57 // process's memory.
58 func (c *Cipher) Reset() {
59     for i := range c.s {
60         c.s[i] = 0
61     }
62     c.i, c.j = 0, 0
63 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/rsa/pkcs1v15.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package rsa
6
7 import (
8     "crypto"
9     "crypto/subtle"
10    "errors"
11    "io"
12    "math/big"
13 )
14
15 // This file implements encryption and decryption using PKCS
16
17 // EncryptPKCS1v15 encrypts the given message with RSA and t
18 // The message must be no longer than the length of the publ
19 // WARNING: use of this function to encrypt plaintexts other
20 // is dangerous. Use RSA OAEP in new protocols.
21 func EncryptPKCS1v15(rand io.Reader, pub *PublicKey, msg []b
22     k := (pub.N.BitLen() + 7) / 8
23     if len(msg) > k-11 {
24         err = ErrorMessageTooLong
25         return
26     }
27
28     // EM = 0x02 || PS || 0x00 || M
29     em := make([]byte, k-1)
30     em[0] = 2
31     ps, mm := em[1:len(em)-len(msg)-1], em[len(em)-len(m
32     err = nonZeroRandomBytes(ps, rand)
33     if err != nil {
34         return
35     }
36     em[len(em)-len(msg)-1] = 0
37     copy(mm, msg)
38
39     m := new(big.Int).SetBytes(em)
40     c := encrypt(new(big.Int), pub, m)
41     out = c.Bytes()
```

```

42         return
43     }
44
45     // DecryptPKCS1v15 decrypts a plaintext using RSA and the pa
46     // If rand != nil, it uses RSA blinding to avoid timing side
47     func DecryptPKCS1v15(rand io.Reader, priv *PrivateKey, ciphe
48         valid, out, err := decryptPKCS1v15(rand, priv, ciphe
49         if err == nil && valid == 0 {
50             err = ErrDecryption
51         }
52
53         return
54     }
55
56     // DecryptPKCS1v15SessionKey decrypts a session key using RS
57     // If rand != nil, it uses RSA blinding to avoid timing side
58     // It returns an error if the ciphertext is the wrong length
59     // ciphertext is greater than the public modulus. Otherwise,
60     // returned. If the padding is valid, the resulting plaintex
61     // into key. Otherwise, key is unchanged. These alternatives
62     // time. It is intended that the user of this function gener
63     // session key beforehand and continue the protocol with the
64     // This will remove any possibility that an attacker can lea
65     // about the plaintext.
66     // See ``Chosen Ciphertext Attacks Against Protocols Based o
67     // Encryption Standard PKCS #1'', Daniel Bleichenbacher, Adv
68     // (Crypto '98).
69     func DecryptPKCS1v15SessionKey(rand io.Reader, priv *Private
70         k := (priv.N.BitLen() + 7) / 8
71         if k-(len(key)+3+8) < 0 {
72             err = ErrDecryption
73             return
74         }
75
76         valid, msg, err := decryptPKCS1v15(rand, priv, ciphe
77         if err != nil {
78             return
79         }
80
81         valid &= subtle.ConstantTimeEq(int32(len(msg)), int3
82         subtle.ConstantTimeCopy(valid, key, msg)
83         return
84     }
85
86     func decryptPKCS1v15(rand io.Reader, priv *PrivateKey, ciphe
87         k := (priv.N.BitLen() + 7) / 8
88         if k < 11 {
89             err = ErrDecryption
90             return
91         }

```

```

92
93     c := new(big.Int).SetBytes(ciphertext)
94     m, err := decrypt(rand, priv, c)
95     if err != nil {
96         return
97     }
98
99     em := leftPad(m.Bytes(), k)
100    firstByteIsZero := subtle.ConstantTimeByteEq(em[0],
101    secondByteIsTwo := subtle.ConstantTimeByteEq(em[1],
102
103    // The remainder of the plaintext must be a string of
104    // octets, followed by a 0, followed by the message.
105    //   lookingForIndex: 1 iff we are still looking for
106    //   index: the offset of the first zero byte.
107    var lookingForIndex, index int
108    lookingForIndex = 1
109
110    for i := 2; i < len(em); i++ {
111        equals0 := subtle.ConstantTimeByteEq(em[i],
112        index = subtle.ConstantTimeSelect(lookingFor
113        lookingForIndex = subtle.ConstantTimeSelect(
114    }
115
116    valid = firstByteIsZero & secondByteIsTwo & (^lookin
117    msg = em[index+1:]
118    return
119 }
120
121 // nonZeroRandomBytes fills the given slice with non-zero ra
122 func nonZeroRandomBytes(s []byte, rand io.Reader) (err error
123     _, err = io.ReadFull(rand, s)
124     if err != nil {
125         return
126     }
127
128     for i := 0; i < len(s); i++ {
129         for s[i] == 0 {
130             _, err = io.ReadFull(rand, s[i:i+1])
131             if err != nil {
132                 return
133             }
134             // In tests, the PRNG may return all
135             // this to break the loop.
136             s[i] ^= 0x42
137         }
138     }
139
140     return

```

```

141 }
142
143 // These are ASN1 DER structures:
144 //   DigestInfo ::= SEQUENCE {
145 //     digestAlgorithm AlgorithmIdentifier,
146 //     digest OCTET STRING
147 //   }
148 // For performance, we don't use the generic ASN1 encoder. R
149 // precompute a prefix of the digest value that makes a vali
150 // with the correct contents.
151 var hashPrefixes = map[crypto.Hash][]byte{
152     crypto.MD5:      {0x30, 0x20, 0x30, 0x0c, 0x06, 0x0
153     crypto.SHA1:     {0x30, 0x21, 0x30, 0x09, 0x06, 0x0
154     crypto.SHA256:   {0x30, 0x31, 0x30, 0x0d, 0x06, 0x0
155     crypto.SHA384:   {0x30, 0x41, 0x30, 0x0d, 0x06, 0x0
156     crypto.SHA512:   {0x30, 0x51, 0x30, 0x0d, 0x06, 0x0
157     crypto.MD5SHA1:  {}, // A special TLS case which do
158     crypto.RIPEMD160: {0x30, 0x20, 0x30, 0x08, 0x06, 0x0
159 }
160
161 // SignPKCS1v15 calculates the signature of hashed using RSA
162 // Note that hashed must be the result of hashing the input
163 // given hash function.
164 func SignPKCS1v15(rand io.Reader, priv *PrivateKey, hash cry
165     hashLen, prefix, err := pkcs1v15HashInfo(hash, len(h
166     if err != nil {
167         return
168     }
169
170     tLen := len(prefix) + hashLen
171     k := (priv.N.BitLen() + 7) / 8
172     if k < tLen+11 {
173         return nil, ErrMessageTooLong
174     }
175
176     // EM = 0x00 || 0x01 || PS || 0x00 || T
177     em := make([]byte, k)
178     em[1] = 1
179     for i := 2; i < k-tLen-1; i++ {
180         em[i] = 0xff
181     }
182     copy(em[k-tLen:k-hashLen], prefix)
183     copy(em[k-hashLen:k], hashed)
184
185     m := new(big.Int).SetBytes(em)
186     c, err := decrypt(rand, priv, m)
187     if err == nil {
188         s = c.Bytes()
189     }

```

```

190         return
191     }
192
193 // VerifyPKCS1v15 verifies an RSA PKCS#1 v1.5 signature.
194 // hashed is the result of hashing the input message using t
195 // function and sig is the signature. A valid signature is i
196 // returning a nil error.
197 func VerifyPKCS1v15(pub *PublicKey, hash crypto.Hash, hashed
198     hashLen, prefix, err := pkcs1v15HashInfo(hash, len(h
199     if err != nil {
200         return
201     }
202
203     tLen := len(prefix) + hashLen
204     k := (pub.N.BitLen() + 7) / 8
205     if k < tLen+11 {
206         err = ErrVerification
207         return
208     }
209
210     c := new(big.Int).SetBytes(sig)
211     m := encrypt(new(big.Int), pub, c)
212     em := leftPad(m.Bytes(), k)
213     // EM = 0x00 || 0x01 || PS || 0x00 || T
214
215     ok := subtle.ConstantTimeByteEq(em[0], 0)
216     ok &= subtle.ConstantTimeByteEq(em[1], 1)
217     ok &= subtle.ConstantTimeCompare(em[k-hashLen:k], ha
218     ok &= subtle.ConstantTimeCompare(em[k-tLen:k-hashLen
219     ok &= subtle.ConstantTimeByteEq(em[k-tLen-1], 0)
220
221     for i := 2; i < k-tLen-1; i++ {
222         ok &= subtle.ConstantTimeByteEq(em[i], 0xff)
223     }
224
225     if ok != 1 {
226         return ErrVerification
227     }
228
229     return nil
230 }
231
232 func pkcs1v15HashInfo(hash crypto.Hash, inLen int) (hashLen
233     hashLen = hash.Size()
234     if inLen != hashLen {
235         return 0, nil, errors.New("input must be has
236     }
237     prefix, ok := hashPrefixes[hash]
238     if !ok {
239         return 0, nil, errors.New("unsupported hash

```

```
240         }
241     return
242 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/crypto/rsa/rsa.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package rsa implements RSA encryption as specified in PKC
6 package rsa
7
8 // TODO(agl): Add support for PSS padding.
9
10 import (
11     "crypto/rand"
12     "crypto/subtle"
13     "errors"
14     "hash"
15     "io"
16     "math/big"
17 )
18
19 var bigZero = big.NewInt(0)
20 var bigOne = big.NewInt(1)
21
22 // A PublicKey represents the public part of an RSA key.
23 type PublicKey struct {
24     N *big.Int // modulus
25     E int      // public exponent
26 }
27
28 // A PrivateKey represents an RSA key
29 type PrivateKey struct {
30     PublicKey // public part.
31     D *big.Int // private exponent
32     Primes []*big.Int // prime factors of N, has >= 2
33
34     // Precomputed contains precomputed values that speed
35     // operations, if available.
36     Precomputed PrecomputedValues
37 }
38
39 type PrecomputedValues struct {
40     Dp, Dq *big.Int // D mod (P-1) (or mod Q-1)
41     Qinv *big.Int // Q^-1 mod Q
42
43     // CRTValues is used for the 3rd and subsequent primes
44     // historical accident, the CRT for the first two primes
```

```

45         // differently in PKCS#1 and interoperability is suf
46         // important that we mirror this.
47         CRTValues []CRTValue
48     }
49
50     // CRTValue contains the precomputed chinese remainder theor
51     type CRTValue struct {
52         Exp    *big.Int // D mod (prime-1).
53         Coeff  *big.Int // R.Coeff ≡ 1 mod Prime.
54         R      *big.Int // product of primes prior to this (i
55     }
56
57     // Validate performs basic sanity checks on the key.
58     // It returns nil if the key is valid, or else an error desc
59     func (priv *PrivateKey) Validate() error {
60         // Check that the prime factors are actually prime.
61         // just a sanity check. Since the random witnesses c
62         // ProbablyPrime are deterministic, given the candid
63         // easy for an attack to generate composites that pa
64         for _, prime := range priv.Primes {
65             if !prime.ProbablyPrime(20) {
66                 return errors.New("prime factor is c
67             }
68         }
69
70         // Check that Πprimes == n.
71         modulus := new(big.Int).Set(bigOne)
72         for _, prime := range priv.Primes {
73             modulus.Mul(modulus, prime)
74         }
75         if modulus.Cmp(priv.N) != 0 {
76             return errors.New("invalid modulus")
77         }
78         // Check that e and totient(Πprimes) are coprime.
79         totient := new(big.Int).Set(bigOne)
80         for _, prime := range priv.Primes {
81             pminus1 := new(big.Int).Sub(prime, bigOne)
82             totient.Mul(totient, pminus1)
83         }
84         e := big.NewInt(int64(priv.E))
85         gcd := new(big.Int)
86         x := new(big.Int)
87         y := new(big.Int)
88         gcd.GCD(x, y, totient, e)
89         if gcd.Cmp(bigOne) != 0 {
90             return errors.New("invalid public exponent E
91         }
92         // Check that de ≡ 1 (mod totient(Πprimes))
93         de := new(big.Int).Mul(priv.D, e)
94         de.Mod(de, totient)

```

```

95         if de.Cmp(bigOne) != 0 {
96             return errors.New("invalid private exponent
97         }
98         return nil
99     }
100
101 // GenerateKey generates an RSA keypair of the given bit siz
102 func GenerateKey(random io.Reader, bits int) (priv *PrivateK
103     return GenerateMultiPrimeKey(random, 2, bits)
104 }
105
106 // GenerateMultiPrimeKey generates a multi-prime RSA keypair
107 // size, as suggested in [1]. Although the public keys are c
108 // (actually, indistinguishable) from the 2-prime case, the
109 // not. Thus it may not be possible to export multi-prime pr
110 // certain formats or to subsequently import them into other
111 //
112 // Table 1 in [2] suggests maximum numbers of primes for a g
113 //
114 // [1] US patent 4405829 (1972, expired)
115 // [2] http://www.cacr.math.uwaterloo.ca/techreports/2006/ca
116 func GenerateMultiPrimeKey(random io.Reader, nprimes int, bi
117     priv = new(PrivateKey)
118     priv.E = 65537
119
120     if nprimes < 2 {
121         return nil, errors.New("rsa.GenerateMultiPri
122     }
123
124     primes := make([]*big.Int, nprimes)
125
126 NextSetOfPrimes:
127     for {
128         todo := bits
129         for i := 0; i < nprimes; i++ {
130             primes[i], err = rand.Prime(random,
131                 if err != nil {
132                     return nil, err
133                 }
134             todo -= primes[i].BitLen()
135         }
136
137         // Make sure that primes is pairwise unequal
138         for i, prime := range primes {
139             for j := 0; j < i; j++ {
140                 if prime.Cmp(primes[j]) == 0
141                     continue NextSetOfPr
142             }
143         }

```

```

144         }
145
146         n := new(big.Int).Set(bigOne)
147         totient := new(big.Int).Set(bigOne)
148         pminus1 := new(big.Int)
149         for _, prime := range primes {
150             n.Mul(n, prime)
151             pminus1.Sub(prime, bigOne)
152             totient.Mul(totient, pminus1)
153         }
154
155         g := new(big.Int)
156         priv.D = new(big.Int)
157         y := new(big.Int)
158         e := big.NewInt(int64(priv.E))
159         g.GCD(priv.D, y, e, totient)
160
161         if g.Cmp(bigOne) == 0 {
162             priv.D.Add(priv.D, totient)
163             priv.Primes = primes
164             priv.N = n
165
166             break
167         }
168     }
169
170     priv.Precompute()
171     return
172 }
173
174 // incCounter increments a four byte, big-endian counter.
175 func incCounter(c *[4]byte) {
176     if c[3]++; c[3] != 0 {
177         return
178     }
179     if c[2]++; c[2] != 0 {
180         return
181     }
182     if c[1]++; c[1] != 0 {
183         return
184     }
185     c[0]++
186 }
187
188 // mgf1XOR XORs the bytes in out with a mask generated using
189 // specified in PKCS#1 v2.1.
190 func mgf1XOR(out []byte, hash hash.Hash, seed []byte) {
191     var counter [4]byte
192     var digest []byte

```

```

193
194     done := 0
195     for done < len(out) {
196         hash.Write(seed)
197         hash.Write(counter[0:4])
198         digest = hash.Sum(digest[:0])
199         hash.Reset()
200
201         for i := 0; i < len(digest) && done < len(ou
202             out[done] ^= digest[i]
203             done++
204         }
205         incCounter(&counter)
206     }
207 }
208
209 // ErrorMessageTooLong is returned when attempting to encrypt
210 // too large for the size of the public key.
211 var ErrorMessageTooLong = errors.New("crypto/rsa: message too
212
213 func encrypt(c *big.Int, pub *PublicKey, m *big.Int) *big.In
214     e := big.NewInt(int64(pub.E))
215     c.Exp(m, e, pub.N)
216     return c
217 }
218
219 // EncryptOAEP encrypts the given message with RSA-OAEP.
220 // The message must be no longer than the length of the publ
221 // twice the hash length plus 2.
222 func EncryptOAEP(hash hash.Hash, random io.Reader, pub *Publ
223     hash.Reset()
224     k := (pub.N.BitLen() + 7) / 8
225     if len(msg) > k-2*hash.Size()-2 {
226         err = ErrorMessageTooLong
227         return
228     }
229
230     hash.Write(label)
231     lHash := hash.Sum(nil)
232     hash.Reset()
233
234     em := make([]byte, k)
235     seed := em[1 : 1+hash.Size()]
236     db := em[1+hash.Size():]
237
238     copy(db[0:hash.Size()], lHash)
239     db[len(db)-len(msg)-1] = 1
240     copy(db[len(db)-len(msg):], msg)
241
242     _, err = io.ReadFull(random, seed)

```

```

243     if err != nil {
244         return
245     }
246
247     mgf1XOR(db, hash, seed)
248     mgf1XOR(seed, hash, db)
249
250     m := new(big.Int)
251     m.SetBytes(em)
252     c := encrypt(new(big.Int), pub, m)
253     out = c.Bytes()
254
255     if len(out) < k {
256         // If the output is too small, we need to let
257         t := make([]byte, k)
258         copy(t[k-len(out):], out)
259         out = t
260     }
261
262     return
263 }
264
265 // ErrDecryption represents a failure to decrypt a message.
266 // It is deliberately vague to avoid adaptive attacks.
267 var ErrDecryption = errors.New("crypto/rsa: decryption error")
268
269 // ErrVerification represents a failure to verify a signature.
270 // It is deliberately vague to avoid adaptive attacks.
271 var ErrVerification = errors.New("crypto/rsa: verification error")
272
273 // modInverse returns ia, the inverse of a in the multiplicative
274 // order n. It requires that a be a member of the group (i.e.
275 func modInverse(a, n *big.Int) (*big.Int, bool) {
276     g := new(big.Int)
277     x := new(big.Int)
278     y := new(big.Int)
279     g.GCD(x, y, a, n)
280     if g.Cmp(bigOne) != 0 {
281         // In this case, a and n aren't coprime and
282         // the inverse. This happens because the value
283         // prime (being the product of two primes) is not
284         // prime.
285         return
286     }
287
288     if x.Cmp(bigOne) < 0 {
289         // 0 is not the multiplicative inverse of an
290         // < 1, then x is negative.
291         x.Add(x, n)

```

```

292     }
293
294     return x, true
295 }
296
297 // Precompute performs some calculations that speed up priva
298 // in the future.
299 func (priv *PrivateKey) Precompute() {
300     if priv.Precomputed.Dp != nil {
301         return
302     }
303
304     priv.Precomputed.Dp = new(big.Int).Sub(priv.Primes[0]
305     priv.Precomputed.Dp.Mod(priv.D, priv.Precomputed.Dp)
306
307     priv.Precomputed.Dq = new(big.Int).Sub(priv.Primes[1]
308     priv.Precomputed.Dq.Mod(priv.D, priv.Precomputed.Dq)
309
310     priv.Precomputed.Qinv = new(big.Int).ModInverse(priv
311
312     r := new(big.Int).Mul(priv.Primes[0], priv.Primes[1]
313     priv.Precomputed.CRTValues = make([]CRTValue, len(pr
314     for i := 2; i < len(priv.Primes); i++ {
315         prime := priv.Primes[i]
316         values := &priv.Precomputed.CRTValues[i-2]
317
318         values.Exp = new(big.Int).Sub(prime, bigOne)
319         values.Exp.Mod(priv.D, values.Exp)
320
321         values.R = new(big.Int).Set(r)
322         values.Coeff = new(big.Int).ModInverse(r, pr
323
324         r.Mul(r, prime)
325     }
326 }
327
328 // decrypt performs an RSA decryption, resulting in a plaint
329 // random source is given, RSA blinding is used.
330 func decrypt(random io.Reader, priv *PrivateKey, c *big.Int)
331     // TODO(agl): can we get away with reusing blinds?
332     if c.Cmp(priv.N) > 0 {
333         err = ErrDecryption
334         return
335     }
336
337     var ir *big.Int
338     if random != nil {
339         // Blinding enabled. Blinding involves multi
340         // Then the decryption operation performs (m

```

```

341 // which equals mr mod n. The factor of r ca
342 // by multiplying by the multiplicative inve
343
344 var r *big.Int
345
346 for {
347     r, err = rand.Int(random, priv.N)
348     if err != nil {
349         return
350     }
351     if r.Cmp(bigZero) == 0 {
352         r = bigOne
353     }
354     var ok bool
355     ir, ok = modInverse(r, priv.N)
356     if ok {
357         break
358     }
359 }
360 bigE := big.NewInt(int64(priv.E))
361 rpowe := new(big.Int).Exp(r, bigE, priv.N)
362 cCopy := new(big.Int).Set(c)
363 cCopy.Mul(cCopy, rpowe)
364 cCopy.Mod(cCopy, priv.N)
365 c = cCopy
366 }
367
368 if priv.Precomputed.Dp == nil {
369     m = new(big.Int).Exp(c, priv.D, priv.N)
370 } else {
371     // We have the precalculated values needed f
372     m = new(big.Int).Exp(c, priv.Precomputed.Dp,
373     m2 := new(big.Int).Exp(c, priv.Precomputed.D
374     m.Sub(m, m2)
375     if m.Sign() < 0 {
376         m.Add(m, priv.Primes[0])
377     }
378     m.Mul(m, priv.Precomputed.Qinv)
379     m.Mod(m, priv.Primes[0])
380     m.Mul(m, priv.Primes[1])
381     m.Add(m, m2)
382
383     for i, values := range priv.Precomputed.CRTV
384         prime := priv.Primes[2+i]
385         m2.Exp(c, values.Exp, prime)
386         m2.Sub(m2, m)
387         m2.Mul(m2, values.Coeff)
388         m2.Mod(m2, prime)
389         if m2.Sign() < 0 {
390             m2.Add(m2, prime)

```

```

391         }
392         m2.Mul(m2, values.R)
393         m.Add(m, m2)
394     }
395 }
396
397 if ir != nil {
398     // Unblind.
399     m.Mul(m, ir)
400     m.Mod(m, priv.N)
401 }
402
403 return
404 }
405
406 // DecryptOAEP decrypts ciphertext using RSA-OAEP.
407 // If random != nil, DecryptOAEP uses RSA blinding to avoid
408 func DecryptOAEP(hash hash.Hash, random io.Reader, priv *Pri
409     k := (priv.N.BitLen() + 7) / 8
410     if len(ciphertext) > k ||
411         k < hash.Size()*2+2 {
412         err = ErrDecryption
413         return
414     }
415
416     c := new(big.Int).SetBytes(ciphertext)
417
418     m, err := decrypt(random, priv, c)
419     if err != nil {
420         return
421     }
422
423     hash.Write(label)
424     lHash := hash.Sum(nil)
425     hash.Reset()
426
427     // Converting the plaintext number to bytes will str
428     // leading zeros so we may have to left pad. We do t
429     // to avoid leaking timing information. (Although we
430     // leak the number of leading zeros. It's not clear
431     // anything about this.)
432     em := leftPad(m.Bytes(), k)
433
434     firstByteIsZero := subtle.ConstantTimeByteEq(em[0],
435
436     seed := em[1 : hash.Size()+1]
437     db := em[hash.Size()+1:]
438
439     mgf1XOR(seed, hash, db)

```

```

440         mgf1XOR(db, hash, seed)
441
442         lHash2 := db[0:hash.Size()]
443
444         // We have to validate the plaintext in constant time
445         // attacks like: J. Manger. A Chosen Ciphertext Attack
446         // Asymmetric Encryption Padding (OAEP) as Standardized
447         // v2.0. In J. Kilian, editor, Advances in Cryptology
448         lHash2Good := subtle.ConstantTimeCompare(lHash, lHash2)
449
450         // The remainder of the plaintext must be zero or more
451         // by 0x01, followed by the message.
452         //   lookingForIndex: 1 iff we are still looking for
453         //   index: the offset of the first 0x01 byte
454         //   invalid: 1 iff we saw a non-zero byte before the
455         var lookingForIndex, index, invalid int
456         lookingForIndex = 1
457         rest := db[hash.Size():]
458
459         for i := 0; i < len(rest); i++ {
460             equals0 := subtle.ConstantTimeByteEq(rest[i], 0)
461             equals1 := subtle.ConstantTimeByteEq(rest[i], 1)
462             index = subtle.ConstantTimeSelect(lookingForIndex, index, i)
463             lookingForIndex = subtle.ConstantTimeSelect(equals0, lookingForIndex, 0)
464             invalid = subtle.ConstantTimeSelect(equals1, invalid, 1)
465         }
466
467         if firstByteIsZero & lHash2Good & !invalid & !lookingForIndex {
468             err = ErrDecryption
469             return
470         }
471
472         msg = rest[index+1:]
473         return
474     }
475
476     // leftPad returns a new slice of length size. The contents
477     // are aligned in the new slice.
478     func leftPad(input []byte, size int) (out []byte) {
479         n := len(input)
480         if n > size {
481             n = size
482         }
483         out = make([]byte, size)
484         copy(out[len(out)-n:], input)
485         return
486     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/sha1/sha1.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package sha1 implements the SHA1 hash algorithm as define
6 package sha1
7
8 import (
9     "crypto"
10    "hash"
11 )
12
13 func init() {
14     crypto.RegisterHash(crypto.SHA1, New)
15 }
16
17 // The size of a SHA1 checksum in bytes.
18 const Size = 20
19
20 // The blocksize of SHA1 in bytes.
21 const BlockSize = 64
22
23 const (
24     _Chunk = 64
25     _Init0 = 0x67452301
26     _Init1 = 0xEFCDAB89
27     _Init2 = 0x98BADCFE
28     _Init3 = 0x10325476
29     _Init4 = 0xC3D2E1F0
30 )
31
32 // digest represents the partial evaluation of a checksum.
33 type digest struct {
34     h    [5]uint32
35     x    [_Chunk]byte
36     nx   int
37     len  uint64
38 }
39
40 func (d *digest) Reset() {
41     d.h[0] = _Init0
```

```

42         d.h[1] = _Init1
43         d.h[2] = _Init2
44         d.h[3] = _Init3
45         d.h[4] = _Init4
46         d.nx = 0
47         d.len = 0
48     }
49
50 // New returns a new hash.Hash computing the SHA1 checksum.
51 func New() hash.Hash {
52     d := new(digest)
53     d.Reset()
54     return d
55 }
56
57 func (d *digest) Size() int { return Size }
58
59 func (d *digest) BlockSize() int { return BlockSize }
60
61 func (d *digest) Write(p []byte) (nn int, err error) {
62     nn = len(p)
63     d.len += uint64(nn)
64     if d.nx > 0 {
65         n := len(p)
66         if n > _Chunk-d.nx {
67             n = _Chunk - d.nx
68         }
69         for i := 0; i < n; i++ {
70             d.x[d.nx+i] = p[i]
71         }
72         d.nx += n
73         if d.nx == _Chunk {
74             _Block(d, d.x[0:])
75             d.nx = 0
76         }
77         p = p[n:]
78     }
79     n := _Block(d, p)
80     p = p[n:]
81     if len(p) > 0 {
82         d.nx = copy(d.x[:], p)
83     }
84     return
85 }
86
87 func (d0 *digest) Sum(in []byte) []byte {
88     // Make a copy of d0 so that caller can keep writing
89     d := *d0
90
91     // Padding. Add a 1 bit and 0 bits until 56 bytes r

```

```

92         len := d.len
93         var tmp [64]byte
94         tmp[0] = 0x80
95         if len%64 < 56 {
96             d.Write(tmp[0 : 56-len%64])
97         } else {
98             d.Write(tmp[0 : 64+56-len%64])
99         }
100
101         // Length in bits.
102         len <<= 3
103         for i := uint(0); i < 8; i++ {
104             tmp[i] = byte(len >> (56 - 8*i))
105         }
106         d.Write(tmp[0:8])
107
108         if d.nx != 0 {
109             panic("d.nx != 0")
110         }
111
112         var digest [Size]byte
113         for i, s := range d.h {
114             digest[i*4] = byte(s >> 24)
115             digest[i*4+1] = byte(s >> 16)
116             digest[i*4+2] = byte(s >> 8)
117             digest[i*4+3] = byte(s)
118         }
119
120         return append(in, digest[:]...)
121     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/sha1/sha1block.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // SHA1 block step.
6 // In its own file so that a faster assembly or C version
7 // can be substituted easily.
8
9 package sha1
10
11 const (
12     _K0 = 0x5A827999
13     _K1 = 0x6ED9EBA1
14     _K2 = 0x8F1BBCDC
15     _K3 = 0xCA62C1D6
16 )
17
18 func _Block(dig *digest, p []byte) int {
19     var w [80]uint32
20
21     n := 0
22     h0, h1, h2, h3, h4 := dig.h[0], dig.h[1], dig.h[2],
23     for len(p) >= _Chunk {
24         // Can interlace the computation of w with t
25         // rounds below if needed for speed.
26         for i := 0; i < 16; i++ {
27             j := i * 4
28             w[i] = uint32(p[j])<<24 | uint32(p[j]
29         }
30         for i := 16; i < 80; i++ {
31             tmp := w[i-3] ^ w[i-8] ^ w[i-14] ^ w
32             w[i] = tmp<<1 | tmp>>(32-1)
33         }
34
35         a, b, c, d, e := h0, h1, h2, h3, h4
36
37         // Each of the four 20-iteration rounds
38         // differs only in the computation of f and
39         // the choice of K (_K0, _K1, etc).
40         for i := 0; i < 20; i++ {
41             f := b&c | (^b)&d
```

```

42             a5 := a<<5 | a>>(32-5)
43             b30 := b<<30 | b>>(32-30)
44             t := a5 + f + e + w[i] + _K0
45             a, b, c, d, e = t, a, b30, c, d
46         }
47     for i := 20; i < 40; i++ {
48         f := b ^ c ^ d
49         a5 := a<<5 | a>>(32-5)
50         b30 := b<<30 | b>>(32-30)
51         t := a5 + f + e + w[i] + _K1
52         a, b, c, d, e = t, a, b30, c, d
53     }
54     for i := 40; i < 60; i++ {
55         f := b&c | b&d | c&d
56         a5 := a<<5 | a>>(32-5)
57         b30 := b<<30 | b>>(32-30)
58         t := a5 + f + e + w[i] + _K2
59         a, b, c, d, e = t, a, b30, c, d
60     }
61     for i := 60; i < 80; i++ {
62         f := b ^ c ^ d
63         a5 := a<<5 | a>>(32-5)
64         b30 := b<<30 | b>>(32-30)
65         t := a5 + f + e + w[i] + _K3
66         a, b, c, d, e = t, a, b30, c, d
67     }
68
69     h0 += a
70     h1 += b
71     h2 += c
72     h3 += d
73     h4 += e
74
75     p = p[_Chunk:]
76     n += _Chunk
77 }
78
79     dig.h[0], dig.h[1], dig.h[2], dig.h[3], dig.h[4] = h
80     return n
81 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/sha256/sha256.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package sha256 implements the SHA224 and SHA256 hash algo
6 // in FIPS 180-2.
7 package sha256
8
9 import (
10     "crypto"
11     "hash"
12 )
13
14 func init() {
15     crypto.RegisterHash(crypto.SHA224, New224)
16     crypto.RegisterHash(crypto.SHA256, New)
17 }
18
19 // The size of a SHA256 checksum in bytes.
20 const Size = 32
21
22 // The size of a SHA224 checksum in bytes.
23 const Size224 = 28
24
25 // The blocksize of SHA256 and SHA224 in bytes.
26 const BlockSize = 64
27
28 const (
29     _Chunk      = 64
30     _Init0      = 0x6A09E667
31     _Init1      = 0xBB67AE85
32     _Init2      = 0x3C6EF372
33     _Init3      = 0xA54FF53A
34     _Init4      = 0x510E527F
35     _Init5      = 0x9B05688C
36     _Init6      = 0x1F83D9AB
37     _Init7      = 0x5BE0CD19
38     _Init0_224 = 0xC1059ED8
39     _Init1_224 = 0x367CD507
40     _Init2_224 = 0x3070DD17
41     _Init3_224 = 0xF70E5939
```

```

42     _Init4_224 = 0xFFC00B31
43     _Init5_224 = 0x68581511
44     _Init6_224 = 0x64F98FA7
45     _Init7_224 = 0xBEFA4FA4
46 )
47
48 // digest represents the partial evaluation of a checksum.
49 type digest struct {
50     h      [8]uint32
51     x      [_Chunk]byte
52     nx     int
53     len    uint64
54     is224 bool // mark if this digest is SHA-224
55 }
56
57 func (d *digest) Reset() {
58     if !d.is224 {
59         d.h[0] = _Init0
60         d.h[1] = _Init1
61         d.h[2] = _Init2
62         d.h[3] = _Init3
63         d.h[4] = _Init4
64         d.h[5] = _Init5
65         d.h[6] = _Init6
66         d.h[7] = _Init7
67     } else {
68         d.h[0] = _Init0_224
69         d.h[1] = _Init1_224
70         d.h[2] = _Init2_224
71         d.h[3] = _Init3_224
72         d.h[4] = _Init4_224
73         d.h[5] = _Init5_224
74         d.h[6] = _Init6_224
75         d.h[7] = _Init7_224
76     }
77     d.nx = 0
78     d.len = 0
79 }
80
81 // New returns a new hash.Hash computing the SHA256 checksum
82 func New() hash.Hash {
83     d := new(digest)
84     d.Reset()
85     return d
86 }
87
88 // New224 returns a new hash.Hash computing the SHA224 checksum
89 func New224() hash.Hash {
90     d := new(digest)
91     d.is224 = true

```

```

92         d.Reset()
93         return d
94     }
95
96     func (d *digest) Size() int {
97         if !d.is224 {
98             return Size
99         }
100        return Size224
101    }
102
103    func (d *digest) BlockSize() int { return BlockSize }
104
105    func (d *digest) Write(p []byte) (nn int, err error) {
106        nn = len(p)
107        d.len += uint64(nn)
108        if d.nx > 0 {
109            n := len(p)
110            if n > _Chunk-d.nx {
111                n = _Chunk - d.nx
112            }
113            for i := 0; i < n; i++ {
114                d.x[d.nx+i] = p[i]
115            }
116            d.nx += n
117            if d.nx == _Chunk {
118                _Block(d, d.x[0:])
119                d.nx = 0
120            }
121            p = p[n:]
122        }
123        n := _Block(d, p)
124        p = p[n:]
125        if len(p) > 0 {
126            d.nx = copy(d.x[:], p)
127        }
128        return
129    }
130
131    func (d0 *digest) Sum(in []byte) []byte {
132        // Make a copy of d0 so that caller can keep writing
133        d := *d0
134
135        // Padding. Add a 1 bit and 0 bits until 56 bytes r
136        len := d.len
137        var tmp [64]byte
138        tmp[0] = 0x80
139        if len%64 < 56 {
140            d.Write(tmp[0 : 56-len%64])

```

```

141     } else {
142         d.Write(tmp[0 : 64+56-len%64])
143     }
144
145     // Length in bits.
146     len <<= 3
147     for i := uint(0); i < 8; i++ {
148         tmp[i] = byte(len >> (56 - 8*i))
149     }
150     d.Write(tmp[0:8])
151
152     if d.nx != 0 {
153         panic("d.nx != 0")
154     }
155
156     h := d.h[:]
157     size := Size
158     if d.is224 {
159         h = d.h[:7]
160         size = Size224
161     }
162
163     var digest [Size]byte
164     for i, s := range h {
165         digest[i*4] = byte(s >> 24)
166         digest[i*4+1] = byte(s >> 16)
167         digest[i*4+2] = byte(s >> 8)
168         digest[i*4+3] = byte(s)
169     }
170
171     return append(in, digest[:size]...)
172 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/sha256/sha256block.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // SHA256 block step.
6 // In its own file so that a faster assembly or C version
7 // can be substituted easily.
8
9 package sha256
10
11 var _K = []uint32{
12     0x428a2f98,
13     0x71374491,
14     0xb5c0fbcf,
15     0xe9b5dba5,
16     0x3956c25b,
17     0x59f111f1,
18     0x923f82a4,
19     0xab1c5ed5,
20     0xd807aa98,
21     0x12835b01,
22     0x243185be,
23     0x550c7dc3,
24     0x72be5d74,
25     0x80deb1fe,
26     0x9bdc06a7,
27     0xc19bf174,
28     0xe49b69c1,
29     0xefbe4786,
30     0x0fc19dc6,
31     0x240ca1cc,
32     0x2de92c6f,
33     0x4a7484aa,
34     0x5cb0a9dc,
35     0x76f988da,
36     0x983e5152,
37     0xa831c66d,
38     0xb00327c8,
39     0xbf597fc7,
40     0xc6e00bf3,
41     0xd5a79147,
```

```

42         0x06ca6351,
43         0x14292967,
44         0x27b70a85,
45         0x2e1b2138,
46         0x4d2c6dfc,
47         0x53380d13,
48         0x650a7354,
49         0x766a0abb,
50         0x81c2c92e,
51         0x92722c85,
52         0xa2bfe8a1,
53         0xa81a664b,
54         0xc24b8b70,
55         0xc76c51a3,
56         0xd192e819,
57         0xd6990624,
58         0xf40e3585,
59         0x106aa070,
60         0x19a4c116,
61         0x1e376c08,
62         0x2748774c,
63         0x34b0bcb5,
64         0x391c0cb3,
65         0x4ed8aa4a,
66         0x5b9cca4f,
67         0x682e6ff3,
68         0x748f82ee,
69         0x78a5636f,
70         0x84c87814,
71         0x8cc70208,
72         0x90befffa,
73         0xa4506ceb,
74         0xbef9a3f7,
75         0xc67178f2,
76     }
77
78     func _Block(dig *digest, p []byte) int {
79         var w [64]uint32
80         n := 0
81         h0, h1, h2, h3, h4, h5, h6, h7 := dig.h[0], dig.h[1]
82         for len(p) >= _Chunk {
83             // Can interlace the computation of w with t
84             // rounds below if needed for speed.
85             for i := 0; i < 16; i++ {
86                 j := i * 4
87                 w[i] = uint32(p[j])<<24 | uint32(p[j]
88             }
89             for i := 16; i < 64; i++ {
90                 t1 := (w[i-2]>>17 | w[i-2]<<(32-17))
91

```

```

92             t2 := (w[i-15]>>7 | w[i-15]<<(32-7))
93
94             w[i] = t1 + w[i-7] + t2 + w[i-16]
95         }
96
97         a, b, c, d, e, f, g, h := h0, h1, h2, h3, h4
98
99         for i := 0; i < 64; i++ {
100             t1 := h + ((e>>6 | e<<(32-6)) ^ (e>>
101
102             t2 := ((a>>2 | a<<(32-2)) ^ (a>>13 |
103
104             h = g
105             g = f
106             f = e
107             e = d + t1
108             d = c
109             c = b
110             b = a
111             a = t1 + t2
112         }
113
114         h0 += a
115         h1 += b
116         h2 += c
117         h3 += d
118         h4 += e
119         h5 += f
120         h6 += g
121         h7 += h
122
123         p = p[_Chunk:]
124         n += _Chunk
125     }
126
127     dig.h[0], dig.h[1], dig.h[2], dig.h[3], dig.h[4], di
128     return n
129 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/sha512/sha512.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package sha512 implements the SHA384 and SHA512 hash algo
6 // in FIPS 180-2.
7 package sha512
8
9 import (
10     "crypto"
11     "hash"
12 )
13
14 func init() {
15     crypto.RegisterHash(crypto.SHA384, New384)
16     crypto.RegisterHash(crypto.SHA512, New)
17 }
18
19 // The size of a SHA512 checksum in bytes.
20 const Size = 64
21
22 // The size of a SHA384 checksum in bytes.
23 const Size384 = 48
24
25 // The blocksize of SHA512 and SHA384 in bytes.
26 const BlockSize = 128
27
28 const (
29     _Chunk      = 128
30     _Init0      = 0x6a09e667f3bcc908
31     _Init1      = 0xbb67ae8584caa73b
32     _Init2      = 0x3c6ef372fe94f82b
33     _Init3      = 0xa54ff53a5f1d36f1
34     _Init4      = 0x510e527fade682d1
35     _Init5      = 0x9b05688c2b3e6c1f
36     _Init6      = 0x1f83d9abfb41bd6b
37     _Init7      = 0x5be0cd19137e2179
38     _Init0_384 = 0xcbbb9d5dc1059ed8
39     _Init1_384 = 0x629a292a367cd507
40     _Init2_384 = 0x9159015a3070dd17
41     _Init3_384 = 0x152fec8f70e5939
```

```

42     _Init4_384 = 0x67332667ffc00b31
43     _Init5_384 = 0x8eb44a8768581511
44     _Init6_384 = 0xdb0c2e0d64f98fa7
45     _Init7_384 = 0x47b5481dbefa4fa4
46 )
47
48 // digest represents the partial evaluation of a checksum.
49 type digest struct {
50     h      [8]uint64
51     x      [_Chunk]byte
52     nx     int
53     len    uint64
54     is384 bool // mark if this digest is SHA-384
55 }
56
57 func (d *digest) Reset() {
58     if !d.is384 {
59         d.h[0] = _Init0
60         d.h[1] = _Init1
61         d.h[2] = _Init2
62         d.h[3] = _Init3
63         d.h[4] = _Init4
64         d.h[5] = _Init5
65         d.h[6] = _Init6
66         d.h[7] = _Init7
67     } else {
68         d.h[0] = _Init0_384
69         d.h[1] = _Init1_384
70         d.h[2] = _Init2_384
71         d.h[3] = _Init3_384
72         d.h[4] = _Init4_384
73         d.h[5] = _Init5_384
74         d.h[6] = _Init6_384
75         d.h[7] = _Init7_384
76     }
77     d.nx = 0
78     d.len = 0
79 }
80
81 // New returns a new hash.Hash computing the SHA512 checksum
82 func New() hash.Hash {
83     d := new(digest)
84     d.Reset()
85     return d
86 }
87
88 // New384 returns a new hash.Hash computing the SHA384 checksum
89 func New384() hash.Hash {
90     d := new(digest)
91     d.is384 = true

```

```

92         d.Reset()
93         return d
94     }
95
96     func (d *digest) Size() int {
97         if !d.is384 {
98             return Size
99         }
100        return Size384
101    }
102
103    func (d *digest) BlockSize() int { return BlockSize }
104
105    func (d *digest) Write(p []byte) (nn int, err error) {
106        nn = len(p)
107        d.len += uint64(nn)
108        if d.nx > 0 {
109            n := len(p)
110            if n > _Chunk-d.nx {
111                n = _Chunk - d.nx
112            }
113            for i := 0; i < n; i++ {
114                d.x[d.nx+i] = p[i]
115            }
116            d.nx += n
117            if d.nx == _Chunk {
118                _Block(d, d.x[0:])
119                d.nx = 0
120            }
121            p = p[n:]
122        }
123        n := _Block(d, p)
124        p = p[n:]
125        if len(p) > 0 {
126            d.nx = copy(d.x[:], p)
127        }
128        return
129    }
130
131    func (d0 *digest) Sum(in []byte) []byte {
132        // Make a copy of d0 so that caller can keep writing
133        d := new(digest)
134        *d = *d0
135
136        // Padding. Add a 1 bit and 0 bits until 112 bytes
137        len := d.len
138        var tmp [128]byte
139        tmp[0] = 0x80
140        if len%128 < 112 {

```

```

141         d.Write(tmp[0 : 112-len%128])
142     } else {
143         d.Write(tmp[0 : 128+112-len%128])
144     }
145
146     // Length in bits.
147     len <= 3
148     for i := uint(0); i < 16; i++ {
149         tmp[i] = byte(len >> (120 - 8*i))
150     }
151     d.Write(tmp[0:16])
152
153     if d.nx != 0 {
154         panic("d.nx != 0")
155     }
156
157     h := d.h[:]
158     size := Size
159     if d.is384 {
160         h = d.h[:6]
161         size = Size384
162     }
163
164     var digest [Size]byte
165     for i, s := range h {
166         digest[i*8] = byte(s >> 56)
167         digest[i*8+1] = byte(s >> 48)
168         digest[i*8+2] = byte(s >> 40)
169         digest[i*8+3] = byte(s >> 32)
170         digest[i*8+4] = byte(s >> 24)
171         digest[i*8+5] = byte(s >> 16)
172         digest[i*8+6] = byte(s >> 8)
173         digest[i*8+7] = byte(s)
174     }
175
176     return append(in, digest[:size]...)
177 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/sha512/sha512block.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // SHA512 block step.
6 // In its own file so that a faster assembly or C version
7 // can be substituted easily.
8
9 package sha512
10
11 var _K = []uint64{
12     0x428a2f98d728ae22,
13     0x7137449123ef65cd,
14     0xb5c0fbcfec4d3b2f,
15     0xe9b5dba58189dbbc,
16     0x3956c25bf348b538,
17     0x59f111f1b605d019,
18     0x923f82a4af194f9b,
19     0xab1c5ed5da6d8118,
20     0xd807aa98a3030242,
21     0x12835b0145706fbe,
22     0x243185be4ee4b28c,
23     0x550c7dc3d5ffb4e2,
24     0x72be5d74f27b896f,
25     0x80deb1fe3b1696b1,
26     0x9bdc06a725c71235,
27     0xc19bf174cf692694,
28     0xe49b69c19ef14ad2,
29     0xefbe4786384f25e3,
30     0x0fc19dc68b8cd5b5,
31     0x240ca1cc77ac9c65,
32     0x2de92c6f592b0275,
33     0x4a7484aa6ea6e483,
34     0x5cb0a9dcbd41fbd4,
35     0x76f988da831153b5,
36     0x983e5152ee66dfab,
37     0xa831c66d2db43210,
38     0xb00327c898fb213f,
39     0xbf597fc7beef0ee4,
40     0xc6e00bf33da88fc2,
41     0xd5a79147930aa725,
```

42 0x06ca6351e003826f,
43 0x142929670a0e6e70,
44 0x27b70a8546d22ffc,
45 0x2e1b21385c26c926,
46 0x4d2c6dfc5ac42aed,
47 0x53380d139d95b3df,
48 0x650a73548baf63de,
49 0x766a0abb3c77b2a8,
50 0x81c2c92e47edae6,
51 0x92722c851482353b,
52 0xa2bfe8a14cf10364,
53 0xa81a664bbc423001,
54 0xc24b8b70d0f89791,
55 0xc76c51a30654be30,
56 0xd192e819d6ef5218,
57 0xd69906245565a910,
58 0xf40e35855771202a,
59 0x106aa07032bbd1b8,
60 0x19a4c116b8d2d0c8,
61 0x1e376c085141ab53,
62 0x2748774cdf8eeb99,
63 0x34b0bcb5e19b48a8,
64 0x391c0cb3c5c95a63,
65 0x4ed8aa4ae3418acb,
66 0x5b9cca4f7763e373,
67 0x682e6ff3d6b2b8a3,
68 0x748f82ee5defb2fc,
69 0x78a5636f43172f60,
70 0x84c87814a1f0ab72,
71 0x8cc702081a6439ec,
72 0x90beffa23631e28,
73 0xa4506cebde82bde9,
74 0xbef9a3f7b2c67915,
75 0xc67178f2e372532b,
76 0xca273ecea26619c,
77 0xd186b8c721c0c207,
78 0xeda7dd6cde0eb1e,
79 0xf57d4f7fee6ed178,
80 0x06f067aa72176fba,
81 0x0a637dc5a2c898a6,
82 0x113f9804bef90dae,
83 0x1b710b35131c471b,
84 0x28db77f523047d84,
85 0x32caab7b40c72493,
86 0x3c9ebe0a15c9becb,
87 0x431d67c49c100d4c,
88 0x4cc5d4becb3e42b6,
89 0x597f299cfc657e2a,
90 0x5fcb6fab3ad6faec,
91 0x6c44198c4a475817,

```

92 }
93
94 func _Block(dig *digest, p []byte) int {
95     var w [80]uint64
96     n := 0
97     h0, h1, h2, h3, h4, h5, h6, h7 := dig.h[0], dig.h[1]
98     for len(p) >= _Chunk {
99         for i := 0; i < 16; i++ {
100             j := i * 8
101             w[i] = uint64(p[j])<<56 | uint64(p[j]
102                 uint64(p[j+4])<<24 | uint64(
103             }
104         for i := 16; i < 80; i++ {
105             t1 := (w[i-2]>>19 | w[i-2]<<(64-19))
106
107             t2 := (w[i-15]>>1 | w[i-15]<<(64-1))
108
109             w[i] = t1 + w[i-7] + t2 + w[i-16]
110         }
111
112         a, b, c, d, e, f, g, h := h0, h1, h2, h3, h4
113
114         for i := 0; i < 80; i++ {
115             t1 := h + ((e>>14 | e<<(64-14)) ^ (e
116
117             t2 := ((a>>28 | a<<(64-28)) ^ (a>>34
118
119             h = g
120             g = f
121             f = e
122             e = d + t1
123             d = c
124             c = b
125             b = a
126             a = t1 + t2
127         }
128
129         h0 += a
130         h1 += b
131         h2 += c
132         h3 += d
133         h4 += e
134         h5 += f
135         h6 += g
136         h7 += h
137
138         p = p[_Chunk:]
139         n += _Chunk
140     }

```

```
141
142     dig.h[0], dig.h[1], dig.h[2], dig.h[3], dig.h[4], di
143     return n
144 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/subtle/constant_time.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package subtle implements functions that are often useful
6 // code but require careful thought to use correctly.
7 package subtle
8
9 // ConstantTimeCompare returns 1 iff the two equal length slices
10 // x and y, have equal contents. The time taken is a function
11 // of the length of the slices and is independent of the contents.
12 func ConstantTimeCompare(x, y []byte) int {
13     var v byte
14
15     for i := 0; i < len(x); i++ {
16         v |= x[i] ^ y[i]
17     }
18
19     return ConstantTimeByteEq(v, 0)
20 }
21
22 // ConstantTimeSelect returns x if v is 1 and y if v is 0.
23 // Its behavior is undefined if v takes any other value.
24 func ConstantTimeSelect(v, x, y int) int { return ^(v-1)&x |
25
26 // ConstantTimeByteEq returns 1 if x == y and 0 otherwise.
27 func ConstantTimeByteEq(x, y uint8) int {
28     z := ^(x ^ y)
29     z &= z >> 4
30     z &= z >> 2
31     z &= z >> 1
32
33     return int(z)
34 }
35
36 // ConstantTimeEq returns 1 if x == y and 0 otherwise.
37 func ConstantTimeEq(x, y int32) int {
38     z := ^(x ^ y)
39     z &= z >> 16
40     z &= z >> 8
41     z &= z >> 4
```

```
42         z &= z >> 2
43         z &= z >> 1
44
45         return int(z & 1)
46     }
47
48     // ConstantTimeCopy copies the contents of y into x iff v ==
49     // Its behavior is undefined if v takes any other value.
50     func ConstantTimeCopy(v int, x, y []byte) {
51         xmask := byte(v - 1)
52         ymask := byte(^v - 1)
53         for i := 0; i < len(x); i++ {
54             x[i] = x[i]&xmask | y[i]&ymask
55         }
56         return
57     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/crypto/tls/alert.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tls
6
7 import "strconv"
8
9 type alert uint8
10
11 const (
12     // alert level
13     alertLevelWarning = 1
14     alertLevelError   = 2
15 )
16
17 const (
18     alertCloseNotify           alert = 0
19     alertUnexpectedMessage    alert = 10
20     alertBadRecordMAC         alert = 20
21     alertDecryptionFailed     alert = 21
22     alertRecordOverflow       alert = 22
23     alertDecompressionFailure alert = 30
24     alertHandshakeFailure     alert = 40
25     alertBadCertificate       alert = 42
26     alertUnsupportedCertificate alert = 43
27     alertCertificateRevoked    alert = 44
28     alertCertificateExpired    alert = 45
29     alertCertificateUnknown    alert = 46
30     alertIllegalParameter     alert = 47
31     alertUnknownCA            alert = 48
32     alertAccessDenied         alert = 49
33     alertDecodeError          alert = 50
34     alertDecryptError         alert = 51
35     alertProtocolVersion      alert = 70
36     alertInsufficientSecurity  alert = 71
37     alertInternalError        alert = 80
38     alertUserCanceled         alert = 90
39     alertNoRenegotiation      alert = 100
40 )
41
42 var alertText = map[alert]string{
43     alertCloseNotify: "close notify",
44     alertUnexpectedMessage: "unexpected message",
```

```

45     alertBadRecordMAC:           "bad record MAC",
46     alertDecryptionFailed:       "decryption failed",
47     alertRecordOverflow:         "record overflow",
48     alertDecompressionFailure:   "decompression failure"
49     alertHandshakeFailure:       "handshake failure",
50     alertBadCertificate:         "bad certificate",
51     alertUnsupportedCertificate: "unsupported certificat
52     alertCertificateRevoked:     "revoked certificate",
53     alertCertificateExpired:     "expired certificate",
54     alertCertificateUnknown:     "unknown certificate",
55     alertIllegalParameter:       "illegal parameter",
56     alertUnknownCA:             "unknown certificate au
57     alertAccessDenied:          "access denied",
58     alertDecodeError:           "error decoding message
59     alertDecryptError:          "error decrypting messa
60     alertProtocolVersion:       "protocol version not s
61     alertInsufficientSecurity:   "insufficient security
62     alertInternalError:         "internal error",
63     alertUserCanceled:         "user canceled",
64     alertNoRenegotiation:       "no renegotiation",
65 }
66
67 func (e alert) String() string {
68     s, ok := alertText[e]
69     if ok {
70         return s
71     }
72     return "alert(" + strconv.Itoa(int(e)) + ")"
73 }
74
75 func (e alert) Error() string {
76     return e.String()
77 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/tls/cipher_suites.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tls
6
7 import (
8     "crypto/aes"
9     "crypto/cipher"
10    "crypto/des"
11    "crypto/hmac"
12    "crypto/rc4"
13    "crypto/sha1"
14    "crypto/x509"
15    "hash"
16 )
17
18 // a keyAgreement implements the client and server side of a
19 // protocol by generating and processing key exchange messages
20 type keyAgreement interface {
21     // On the server side, the first two methods are called
22
23     // In the case that the key agreement protocol doesn't
24     // ServerKeyExchange message, generateServerKeyExchange
25     // nil.
26     generateServerKeyExchange(*Config, *Certificate, *ClientHelloMsg)
27     processClientKeyExchange(*Config, *Certificate, *ClientHelloMsg)
28
29     // On the client side, the next two methods are called
30
31     // This method may not be called if the server doesn't
32     // ServerKeyExchange message.
33     processServerKeyExchange(*Config, *ClientHelloMsg, *ServerKeyExchangeMsg)
34     generateClientKeyExchange(*Config, *ClientHelloMsg, *ServerKeyExchangeMsg)
35 }
36
37 // A cipherSuite is a specific combination of key agreement,
38 // cipher, and MAC function. All cipher suites currently assume RSA key agreement
39 type cipherSuite struct {
40     id uint16
41     // the lengths, in bytes, of the key material needed
```

```

42     keyLen int
43     macLen int
44     ivLen  int
45     ka     func() keyAgreement
46     // If elliptic is set, a server will only consider t
47     // the ClientHello indicated that the client support
48     // and point format that we can handle.
49     elliptic bool
50     cipher func(key, iv []byte, isRead bool) interface
51     mac     func(version uint16, macKey []byte) macFunc
52 }
53
54 var cipherSuites = []*cipherSuite{
55     {TLS_RSA_WITH_RC4_128_SHA, 16, 20, 0, rsaKA, false,
56     {TLS_RSA_WITH_3DES_EDE_CBC_SHA, 24, 20, 8, rsaKA, fa
57     {TLS_RSA_WITH_AES_128_CBC_SHA, 16, 20, 16, rsaKA, fa
58     {TLS_ECDHE_RSA_WITH_RC4_128_SHA, 16, 20, 0, ecdheRSA
59     {TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA, 24, 20, 8, ecd
60     {TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA, 16, 20, 16, ecd
61 }
62
63 func cipherRC4(key, iv []byte, isRead bool) interface{} {
64     cipher, _ := rc4.NewCipher(key)
65     return cipher
66 }
67
68 func cipher3DES(key, iv []byte, isRead bool) interface{} {
69     block, _ := des.NewTripleDESCipher(key)
70     if isRead {
71         return cipher.NewCBCDecrypter(block, iv)
72     }
73     return cipher.NewCBCEncrypter(block, iv)
74 }
75
76 func cipherAES(key, iv []byte, isRead bool) interface{} {
77     block, _ := aes.NewCipher(key)
78     if isRead {
79         return cipher.NewCBCDecrypter(block, iv)
80     }
81     return cipher.NewCBCEncrypter(block, iv)
82 }
83
84 // macSHA1 returns a macFunction for the given protocol vers
85 func macSHA1(version uint16, key []byte) macFunction {
86     if version == versionSSL30 {
87         mac := ssl30MAC{
88             h: sha1.New(),
89             key: make([]byte, len(key)),
90         }
91         copy(mac.key, key)

```

```

92         return mac
93     }
94     return tls10MAC{hmac.New(sha1.New, key)}
95 }
96
97 type macFunction interface {
98     Size() int
99     MAC(digestBuf, seq, data []byte) []byte
100 }
101
102 // ssl30MAC implements the SSLv3 MAC function, as defined in
103 // www.mozilla.org/projects/security/pki/nss/ssl/draft302.tx
104 type ssl30MAC struct {
105     h    hash.Hash
106     key []byte
107 }
108
109 func (s ssl30MAC) Size() int {
110     return s.h.Size()
111 }
112
113 var ssl30Pad1 = [48]byte{0x36, 0x36, 0x36, 0x36, 0x36, 0x36,
114
115 var ssl30Pad2 = [48]byte{0x5c, 0x5c, 0x5c, 0x5c, 0x5c, 0x5c,
116
117 func (s ssl30MAC) MAC(digestBuf, seq, record []byte) []byte
118     padLength := 48
119     if s.h.Size() == 20 {
120         padLength = 40
121     }
122
123     s.h.Reset()
124     s.h.Write(s.key)
125     s.h.Write(ssl30Pad1[:padLength])
126     s.h.Write(seq)
127     s.h.Write(record[:1])
128     s.h.Write(record[3:5])
129     s.h.Write(record[recordHeaderLen:])
130     digestBuf = s.h.Sum(digestBuf[:0])
131
132     s.h.Reset()
133     s.h.Write(s.key)
134     s.h.Write(ssl30Pad2[:padLength])
135     s.h.Write(digestBuf)
136     return s.h.Sum(digestBuf[:0])
137 }
138
139 // tls10MAC implements the TLS 1.0 MAC function. RFC 2246, s
140 type tls10MAC struct {

```

```

141         h hash.Hash
142     }
143
144     func (s tls10MAC) Size() int {
145         return s.h.Size()
146     }
147
148     func (s tls10MAC) MAC(digestBuf, seq, record []byte) []byte
149         s.h.Reset()
150         s.h.Write(seq)
151         s.h.Write(record)
152         return s.h.Sum(digestBuf[:0])
153     }
154
155     func rsaKA() keyAgreement {
156         return rsaKeyAgreement{}
157     }
158
159     func ecdheRSAKA() keyAgreement {
160         return new(ecdheRSAKeyAgreement)
161     }
162
163     // mutualCipherSuite returns a cipherSuite given a list of s
164     // ciphersuites and the id requested by the peer.
165     func mutualCipherSuite(have []uint16, want uint16) *cipherSu
166         for _, id := range have {
167             if id == want {
168                 for _, suite := range cipherSuites {
169                     if suite.id == want {
170                         return suite
171                     }
172                 }
173                 return nil
174             }
175         }
176         return nil
177     }
178
179     // A list of the possible cipher suite ids. Taken from
180     // http://www.iana.org/assignments/tls-parameters/tls-parame
181     const (
182         TLS_RSA_WITH_RC4_128_SHA           uint16 = 0x0005
183         TLS_RSA_WITH_3DES_EDE_CBC_SHA      uint16 = 0x000a
184         TLS_RSA_WITH_AES_128_CBC_SHA       uint16 = 0x002f
185         TLS_ECDHE_RSA_WITH_RC4_128_SHA     uint16 = 0xc011
186         TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA uint16 = 0xc012
187         TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA uint16 = 0xc013
188     )

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/tls/common.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tls
6
7 import (
8     "crypto"
9     "crypto/rand"
10    "crypto/x509"
11    "io"
12    "strings"
13    "sync"
14    "time"
15 )
16
17 const (
18     maxPlaintext      = 16384           // maximum plaintext
19     maxCiphertext     = 16384 + 2048 // maximum ciphertext
20     recordHeaderLen   = 5              // record header length
21     maxHandshake     = 65536          // maximum handshake
22
23     versionSSL30 = 0x0300
24     versionTLS10 = 0x0301
25
26     minVersion = versionSSL30
27     maxVersion = versionTLS10
28 )
29
30 // TLS record types.
31 type recordType uint8
32
33 const (
34     recordTypeChangeCipherSpec recordType = 20
35     recordTypeAlert              recordType = 21
36     recordTypeHandshake          recordType = 22
37     recordTypeApplicationData    recordType = 23
38 )
39
40 // TLS handshake message types.
41 const (
```

```

42         typeClientHello          uint8 = 1
43         typeServerHello          uint8 = 2
44         typeCertificate           uint8 = 11
45         typeServerKeyExchange    uint8 = 12
46         typeCertificateRequest    uint8 = 13
47         typeServerHelloDone      uint8 = 14
48         typeCertificateVerify     uint8 = 15
49         typeClientKeyExchange    uint8 = 16
50         typeFinished              uint8 = 20
51         typeCertificateStatus     uint8 = 22
52         typeNextProtocol          uint8 = 67 // Not IANA assign
53     )
54
55     // TLS compression types.
56     const (
57         compressionNone uint8 = 0
58     )
59
60     // TLS extension numbers
61     var (
62         extensionServerName          uint16 = 0
63         extensionStatusRequest       uint16 = 5
64         extensionSupportedCurves    uint16 = 10
65         extensionSupportedPoints     uint16 = 11
66         extensionNextProtoNeg       uint16 = 13172 // not IANA
67     )
68
69     // TLS Elliptic Curves
70     // http://www.iana.org/assignments/tls-parameters/tls-parame
71     var (
72         curveP256 uint16 = 23
73         curveP384 uint16 = 24
74         curveP521 uint16 = 25
75     )
76
77     // TLS Elliptic Curve Point Formats
78     // http://www.iana.org/assignments/tls-parameters/tls-parame
79     var (
80         pointFormatUncompressed uint8 = 0
81     )
82
83     // TLS CertificateStatusType (RFC 3546)
84     const (
85         statusTypeOCSP uint8 = 1
86     )
87
88     // Certificate types (for certificateRequestMsg)
89     const (
90         certTypeRSASign      = 1 // A certificate containing a
91         certTypeDSSSign     = 2 // A certificate containing a

```

```

92         certTypeRSAFixedDH = 3 // A certificate containing a
93         certTypeDSSFfixedDH = 4 // A certificate containing a
94         // Rest of these are reserved by the TLS spec
95     )
96
97     // ConnectionState records basic TLS details about the connection
98     type ConnectionState struct {
99         HandshakeComplete          bool
100        CipherSuite                 uint16
101        NegotiatedProtocol          string
102        NegotiatedProtocolIsMutual bool
103
104        // ServerName contains the server name indicated by
105        // (Only valid for server connections.)
106        ServerName string
107
108        // the certificate chain that was presented by the client
109        PeerCertificates []*x509.Certificate
110        // the verified certificate chains built from PeerCertificates
111        VerifiedChains [][]*x509.Certificate
112    }
113
114    // ClientAuthType declares the policy the server will follow
115    // TLS Client Authentication.
116    type ClientAuthType int
117
118    const (
119        NoClientCert ClientAuthType = iota
120        RequestClientCert
121        RequireAnyClientCert
122        VerifyClientCertIfGiven
123        RequireAndVerifyClientCert
124    )
125
126    // A Config structure is used to configure a TLS client or server
127    // has been passed to a TLS function it must not be modified
128    type Config struct {
129        // Rand provides the source of entropy for nonces and
130        // If Rand is nil, TLS uses the cryptographic random
131        // crypto/rand.
132        Rand io.Reader
133
134        // Time returns the current time as the number of seconds since
135        // If Time is nil, TLS uses time.Now.
136        Time func() time.Time
137
138        // Certificates contains one or more certificate chains
139        // to present to the other side of the connection.
140        // Server configurations must include at least one certificate

```

```

141     Certificates []Certificate
142
143     // NameToCertificate maps from a certificate name to
144     // Certificates. Note that a certificate name can be
145     // '*.example.com' and so doesn't have to be a domain
146     // See Config.BuildNameToCertificate
147     // The nil value causes the first element of Certificates
148     // for all connections.
149     NameToCertificate map[string]*Certificate
150
151     // RootCAs defines the set of root certificate authorities
152     // that clients use when verifying server certificates.
153     // If RootCAs is nil, TLS uses the host's root CA set.
154     RootCAs *x509.CertPool
155
156     // NextProtos is a list of supported, application level
157     // protocols.
158     NextProtos []string
159
160     // ServerName is included in the client's handshake
161     // to identify the host.
162     ServerName string
163
164     // ClientAuth determines the server's policy for
165     // TLS Client Authentication. The default is NoClientAuth.
166     ClientAuth ClientAuthType
167
168     // ClientCAs defines the set of root certificate authorities
169     // that servers use if required to verify a client's
170     // certificate.
171     ClientCAs *x509.CertPool
172
173     // InsecureSkipVerify controls whether a client verifies the
174     // server's certificate chain and host name.
175     // If InsecureSkipVerify is true, TLS accepts any certificate
176     // presented by the server and any host name in that
177     // certificate's common names or subject alt names.
178     // This should be used only for testing.
179     InsecureSkipVerify bool
180
181     // CipherSuites is a list of supported cipher suites.
182     // If CipherSuites is nil, TLS uses a list of suites supported by
183     // the implementation.
184     CipherSuites []uint16
185 }
186
187 func (c *Config) rand() io.Reader {
188     r := c.Rand
189     if r == nil {
190         return rand.Reader
191     }
192 }

```

```

190         return r
191     }
192
193     func (c *Config) time() time.Time {
194         t := c.Time
195         if t == nil {
196             t = time.Now
197         }
198         return t()
199     }
200
201     func (c *Config) cipherSuites() []uint16 {
202         s := c.CipherSuites
203         if s == nil {
204             s = defaultCipherSuites()
205         }
206         return s
207     }
208
209     // getCertificateForName returns the best certificate for th
210     // defaulting to the first element of c.Certificates if ther
211     // options.
212     func (c *Config) getCertificateForName(name string) *Certifi
213         if len(c.Certificates) == 1 || c.NameToCertificate =
214             // There's only one choice, so no point doin
215             return &c.Certificates[0]
216     }
217
218     name = strings.ToLower(name)
219     for len(name) > 0 && name[len(name)-1] == '.' {
220         name = name[:len(name)-1]
221     }
222
223     if cert, ok := c.NameToCertificate[name]; ok {
224         return cert
225     }
226
227     // try replacing labels in the name with wildcards u
228     // match.
229     labels := strings.Split(name, ".")
230     for i := range labels {
231         labels[i] = "*"
232         candidate := strings.Join(labels, ".")
233         if cert, ok := c.NameToCertificate[candidate]
234             return cert
235     }
236 }
237
238 // If nothing matches, return the first certificate.
239 return &c.Certificates[0]

```

```

240 }
241
242 // BuildNameToCertificate parses c.Certificates and builds c
243 // from the CommonName and SubjectAlternateName fields of ea
244 // certificates.
245 func (c *Config) BuildNameToCertificate() {
246     c.NameToCertificate = make(map[string]*Certificate)
247     for i := range c.Certificates {
248         cert := &c.Certificates[i]
249         x509Cert, err := x509.ParseCertificate(cert.
250             if err != nil {
251                 continue
252             }
253             if len(x509Cert.Subject.CommonName) > 0 {
254                 c.NameToCertificate[x509Cert.Subject
255             }
256             for _, san := range x509Cert.DNSNames {
257                 c.NameToCertificate[san] = cert
258             }
259     }
260 }
261
262 // A Certificate is a chain of one or more certificates, lea
263 type Certificate struct {
264     Certificate [][]byte
265     PrivateKey crypto.PrivateKey // supported types: *r
266     // OCSPStaple contains an optional OCSP response whi
267     // to clients that request it.
268     OCSPStaple []byte
269     // Leaf is the parsed form of the leaf certificate,
270     // initialized using x509.ParseCertificate to reduce
271     // processing for TLS clients doing client authentic
272     // leaf certificate will be parsed as needed.
273     Leaf *x509.Certificate
274 }
275
276 // A TLS record.
277 type record struct {
278     contentType recordType
279     major, minor uint8
280     payload      []byte
281 }
282
283 type handshakeMessage interface {
284     marshal() []byte
285     unmarshal([]byte) bool
286 }
287
288 // mutualVersion returns the protocol version to use given t

```

```

289 // version of the peer.
290 func mutualVersion(vers uint16) (uint16, bool) {
291     if vers < minVersion {
292         return 0, false
293     }
294     if vers > maxVersion {
295         vers = maxVersion
296     }
297     return vers, true
298 }
299
300 var emptyConfig Config
301
302 func defaultConfig() *Config {
303     return &emptyConfig
304 }
305
306 var (
307     once sync.Once
308     varDefaultCipherSuites []uint16
309 )
310
311 func defaultCipherSuites() []uint16 {
312     once.Do(initDefaultCipherSuites)
313     return varDefaultCipherSuites
314 }
315
316 func initDefaultCipherSuites() {
317     varDefaultCipherSuites = make([]uint16, len(cipherSu
318     for i, suite := range cipherSuites {
319         varDefaultCipherSuites[i] = suite.id
320     }
321 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/crypto/tls/conn.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // TLS low level connection and record layer
6
7 package tls
8
9 import (
10     "bytes"
11     "crypto/cipher"
12     "crypto/subtle"
13     "crypto/x509"
14     "errors"
15     "io"
16     "net"
17     "sync"
18     "time"
19 )
20
21 // A Conn represents a secured connection.
22 // It implements the net.Conn interface.
23 type Conn struct {
24     // constant
25     conn net.Conn
26     isClient bool
27
28     // constant after handshake; protected by handshakeM
29     handshakeMutex sync.Mutex // handshakeMutex < in.
30     vers uint16 // TLS version
31     haveVers bool // version has been neg
32     config *Config // configuration passed
33     handshakeComplete bool
34     cipherSuite uint16
35     ocsponse []byte // stapled OCSP response
36     peerCertificates []*x509.Certificate
37     // verifiedChains contains the certificate chains th
38     // opposed to the ones presented by the server.
39     verifiedChains [][]*x509.Certificate
40     // serverName contains the server name indicated by
41     serverName string
42
43     clientProtocol string
44     clientProtocolFallback bool
```

```

45
46     // first permanent error
47     errMutex sync.Mutex
48     err      error
49
50     // input/output
51     in, out  halfConn    // in.Mutex < out.Mutex
52     rawInput *block     // raw input, right off the wi
53     input   *block     // application data waiting to
54     hand    bytes.Buffer // handshake data waiting to b
55
56     tmp [16]byte
57 }
58
59 func (c *Conn) setError(err error) error {
60     c.errMutex.Lock()
61     defer c.errMutex.Unlock()
62
63     if c.err == nil {
64         c.err = err
65     }
66     return err
67 }
68
69 func (c *Conn) error() error {
70     c.errMutex.Lock()
71     defer c.errMutex.Unlock()
72
73     return c.err
74 }
75
76 // Access to net.Conn methods.
77 // Cannot just embed net.Conn because that would
78 // export the struct field too.
79
80 // LocalAddr returns the local network address.
81 func (c *Conn) LocalAddr() net.Addr {
82     return c.conn.LocalAddr()
83 }
84
85 // RemoteAddr returns the remote network address.
86 func (c *Conn) RemoteAddr() net.Addr {
87     return c.conn.RemoteAddr()
88 }
89
90 // SetDeadline sets the read and write deadlines associated
91 // A zero value for t means Read and Write will not time out
92 // After a Write has timed out, the TLS state is corrupt and
93 func (c *Conn) SetDeadline(t time.Time) error {
94     return c.conn.SetDeadline(t)

```

```

95 }
96
97 // SetReadDeadline sets the read deadline on the underlying
98 // A zero value for t means Read will not time out.
99 func (c *Conn) SetReadDeadline(t time.Time) error {
100     return c.conn.SetReadDeadline(t)
101 }
102
103 // SetWriteDeadline sets the write deadline on the underlying
104 // A zero value for t means Write will not time out.
105 // After a Write has timed out, the TLS state is corrupt and
106 func (c *Conn) SetWriteDeadline(t time.Time) error {
107     return c.conn.SetWriteDeadline(t)
108 }
109
110 // A halfConn represents one direction of the record layer
111 // connection, either sending or receiving.
112 type halfConn struct {
113     sync.Mutex
114     version uint16 // protocol version
115     cipher interface{} // cipher algorithm
116     mac macFunction
117     seq [8]byte // 64-bit sequence number
118     bfree *block // list of free blocks
119
120     nextCipher interface{} // next encryption state
121     nextMac macFunction // next MAC algorithm
122
123     // used to save allocating a new buffer for each MAC
124     inDigestBuf, outDigestBuf []byte
125 }
126
127 // prepareCipherSpec sets the encryption and MAC states
128 // that a subsequent changeCipherSpec will use.
129 func (hc *halfConn) prepareCipherSpec(version uint16, cipher
130     hc.version = version
131     hc.nextCipher = cipher
132     hc.nextMac = mac
133 }
134
135 // changeCipherSpec changes the encryption and MAC states
136 // to the ones previously passed to prepareCipherSpec.
137 func (hc *halfConn) changeCipherSpec() error {
138     if hc.nextCipher == nil {
139         return alertInternalError
140     }
141     hc.cipher = hc.nextCipher
142     hc.mac = hc.nextMac
143     hc.nextCipher = nil

```

```

144         hc.nextMac = nil
145         return nil
146     }
147
148     // incSeq increments the sequence number.
149     func (hc *halfConn) incSeq() {
150         for i := 7; i >= 0; i-- {
151             hc.seq[i]++
152             if hc.seq[i] != 0 {
153                 return
154             }
155         }
156
157         // Not allowed to let sequence number wrap.
158         // Instead, must renegotiate before it does.
159         // Not likely enough to bother.
160         panic("TLS: sequence number wraparound")
161     }
162
163     // resetSeq resets the sequence number to zero.
164     func (hc *halfConn) resetSeq() {
165         for i := range hc.seq {
166             hc.seq[i] = 0
167         }
168     }
169
170     // removePadding returns an unpadded slice, in constant time
171     // of the input. It also returns a byte which is equal to 25
172     // was valid and 0 otherwise. See RFC 2246, section 6.2.3.2
173     func removePadding(payload []byte) ([]byte, byte) {
174         if len(payload) < 1 {
175             return payload, 0
176         }
177
178         paddingLen := payload[len(payload)-1]
179         t := uint(len(payload)-1) - uint(paddingLen)
180         // if len(payload) >= (paddingLen - 1) then the MSB
181         good := byte(int32(^t) >> 31)
182
183         toCheck := 255 // the maximum possible padding length
184         // The length of the padded data is public, so we can
185         if toCheck+1 > len(payload) {
186             toCheck = len(payload) - 1
187         }
188
189         for i := 0; i < toCheck; i++ {
190             t := uint(paddingLen) - uint(i)
191             // if i <= paddingLen then the MSB of t is z
192             mask := byte(int32(^t) >> 31)

```

```

193             b := payload[len(payload)-1-i]
194             good ^= mask&paddingLen ^ mask&b
195         }
196
197         // We AND together the bits of good and replicate th
198         // all the bits.
199         good &= good << 4
200         good &= good << 2
201         good &= good << 1
202         good = uint8(int8(good) >> 7)
203
204         toRemove := good&paddingLen + 1
205         return payload[:len(payload)-int(toRemove)], good
206     }
207
208     // removePaddingSSL30 is a replacement for removePadding in
209     // protocol version is SSLv3. In this version, the contents
210     // are random and cannot be checked.
211     func removePaddingSSL30(payload []byte) ([]byte, byte) {
212         if len(payload) < 1 {
213             return payload, 0
214         }
215
216         paddingLen := int(payload[len(payload)-1]) + 1
217         if paddingLen > len(payload) {
218             return payload, 0
219         }
220
221         return payload[:len(payload)-paddingLen], 255
222     }
223
224     func roundUp(a, b int) int {
225         return a + (b-a%b)%b
226     }
227
228     // decrypt checks and strips the mac and decrypts the data i
229     func (hc *halfConn) decrypt(b *block) (bool, alert) {
230         // pull out payload
231         payload := b.data[recordHeaderLen:]
232
233         macSize := 0
234         if hc.mac != nil {
235             macSize = hc.mac.Size()
236         }
237
238         paddingGood := byte(255)
239
240         // decrypt
241         if hc.cipher != nil {
242             switch c := hc.cipher.(type) {

```

```

243     case cipher.Stream:
244         c.XORKeyStream(payload, payload)
245     case cipher.BlockMode:
246         blockSize := c.BlockSize()
247
248         if len(payload)%blockSize != 0 || le
249             return false, alertBadRecord
250     }
251
252     c.CryptBlocks(payload, payload)
253     if hc.version == versionSSL30 {
254         payload, paddingGood = remov
255     } else {
256         payload, paddingGood = remov
257     }
258     b.resize(recordHeaderLen + len(payload)
259
260     // note that we still have a timing
261     // MAC check, below. An attacker can
262     // so that a correct padding will ca
263     // block to be calculated. Then they
264     // decrypt a record by breaking each
265     // "Password Interception in a SSL/T
266     // Canvel et al.
267     //
268     // However, our behavior matches Ope
269     // only as much as they do.
270     default:
271         panic("unknown cipher type")
272     }
273 }
274
275 // check, strip mac
276 if hc.mac != nil {
277     if len(payload) < macSize {
278         return false, alertBadRecordMAC
279     }
280
281     // strip mac off payload, b.data
282     n := len(payload) - macSize
283     b.data[3] = byte(n >> 8)
284     b.data[4] = byte(n)
285     b.resize(recordHeaderLen + n)
286     remoteMAC := payload[n:]
287     localMAC := hc.mac.MAC(hc.inDigestBuf, hc.se
288     hc.incSeq())
289
290     if subtle.ConstantTimeCompare(localMAC, remo
291         return false, alertBadRecordMAC

```

```

292         }
293         hc.inDigestBuf = localMAC
294     }
295
296     return true, 0
297 }
298
299 // padToBlockSize calculates the needed padding block, if an
300 // On exit, prefix aliases payload and extends to the end of
301 // block of payload. finalBlock is a fresh slice which conta
302 // any suffix of payload as well as the needed padding to ma
303 // full block.
304 func padToBlockSize(payload []byte, blockSize int) (prefix,
305     overrun := len(payload) % blockSize
306     paddingLen := blockSize - overrun
307     prefix = payload[:len(payload)-overrun]
308     finalBlock = make([]byte, blockSize)
309     copy(finalBlock, payload[len(payload)-overrun:])
310     for i := overrun; i < blockSize; i++ {
311         finalBlock[i] = byte(paddingLen - 1)
312     }
313     return
314 }
315
316 // encrypt encrypts and macs the data in b.
317 func (hc *halfConn) encrypt(b *block) (bool, alert) {
318     // mac
319     if hc.mac != nil {
320         mac := hc.mac.MAC(hc.outDigestBuf, hc.seq[0:
321             hc.incSeq()
322
323         n := len(b.data)
324         b.resize(n + len(mac))
325         copy(b.data[n:], mac)
326         hc.outDigestBuf = mac
327     }
328
329     payload := b.data[recordHeaderLen:]
330
331     // encrypt
332     if hc.cipher != nil {
333         switch c := hc.cipher.(type) {
334         case cipher.Stream:
335             c.XORKeyStream(payload, payload)
336         case cipher.BlockMode:
337             prefix, finalBlock := padToBlockSize
338             b.resize(recordHeaderLen + len(prefi
339             c.CryptBlocks(b.data[recordHeaderLen
340             c.CryptBlocks(b.data[recordHeaderLen

```

```

341         default:
342             panic("unknown cipher type")
343     }
344 }
345
346 // update length to include MAC and any block paddin
347 n := len(b.data) - recordHeaderLen
348 b.data[3] = byte(n >> 8)
349 b.data[4] = byte(n)
350
351 return true, 0
352 }
353
354 // A block is a simple data buffer.
355 type block struct {
356     data []byte
357     off  int // index for Read
358     link *block
359 }
360
361 // resize resizes block to be n bytes, growing if necessary.
362 func (b *block) resize(n int) {
363     if n > cap(b.data) {
364         b.reserve(n)
365     }
366     b.data = b.data[0:n]
367 }
368
369 // reserve makes sure that block contains a capacity of at l
370 func (b *block) reserve(n int) {
371     if cap(b.data) >= n {
372         return
373     }
374     m := cap(b.data)
375     if m == 0 {
376         m = 1024
377     }
378     for m < n {
379         m *= 2
380     }
381     data := make([]byte, len(b.data), m)
382     copy(data, b.data)
383     b.data = data
384 }
385
386 // readFromUntil reads from r into b until b contains at lea
387 // or else returns an error.
388 func (b *block) readFromUntil(r io.Reader, n int) error {
389     // quick case
390     if len(b.data) >= n {

```

```

391         return nil
392     }
393
394     // read until have enough.
395     b.reserve(n)
396     for {
397         m, err := r.Read(b.data[len(b.data):cap(b.da
398         b.data = b.data[0 : len(b.data)+m]
399         if len(b.data) >= n {
400             break
401         }
402         if err != nil {
403             return err
404         }
405     }
406     return nil
407 }
408
409 func (b *block) Read(p []byte) (n int, err error) {
410     n = copy(p, b.data[b.off:])
411     b.off += n
412     return
413 }
414
415 // newBlock allocates a new block, from hc's free list if po
416 func (hc *halfConn) newBlock() *block {
417     b := hc.bfree
418     if b == nil {
419         return new(block)
420     }
421     hc.bfree = b.link
422     b.link = nil
423     b.resize(0)
424     return b
425 }
426
427 // freeBlock returns a block to hc's free list.
428 // The protocol is such that each side only has a block or t
429 // its free list at a time, so there's no need to worry abou
430 // trimming the list, etc.
431 func (hc *halfConn) freeBlock(b *block) {
432     b.link = hc.bfree
433     hc.bfree = b
434 }
435
436 // splitBlock splits a block after the first n bytes,
437 // returning a block with those n bytes and a
438 // block with the remainder. the latter may be nil.
439 func (hc *halfConn) splitBlock(b *block, n int) (*block, *bl

```

```

440         if len(b.data) <= n {
441             return b, nil
442         }
443         bb := hc.newBlock()
444         bb.resize(len(b.data) - n)
445         copy(bb.data, b.data[n:])
446         b.data = b.data[0:n]
447         return b, bb
448     }
449
450     // readRecord reads the next TLS record from the connection
451     // and updates the record layer state.
452     // c.in.Mutex <= L; c.input == nil.
453     func (c *Conn) readRecord(want recordType) error {
454         // Caller must be in sync with connection:
455         // handshake data if handshake not yet completed,
456         // else application data. (We don't support renegot
457         switch want {
458         default:
459             return c.sendAlert(alertInternalError)
460         case recordTypeHandshake, recordTypeChangeCipherSpec
461             if c.handshakeComplete {
462                 return c.sendAlert(alertInternalError)
463             }
464         case recordTypeApplicationData:
465             if !c.handshakeComplete {
466                 return c.sendAlert(alertInternalError)
467             }
468         }
469
470     Again:
471         if c.rawInput == nil {
472             c.rawInput = c.in.newBlock()
473         }
474         b := c.rawInput
475
476         // Read header, payload.
477         if err := b.readFromUntil(c.conn, recordHeaderLen);
478             // RFC suggests that EOF without an alertClo
479             // an error, but popular web sites seem to d
480             // so we can't make it an error.
481             // if err == io.EOF {
482             //     err = io.ErrUnexpectedEOF
483             // }
484             if e, ok := err.(net.Error); !ok || !e.Tempo
485                 c.setError(err)
486         }
487         return err
488     }

```

```

489     typ := recordType(b.data[0])
490     vers := uint16(b.data[1])<<8 | uint16(b.data[2])
491     n := int(b.data[3])<<8 | int(b.data[4])
492     if c.haveVers && vers != c.vers {
493         return c.sendAlert(alertProtocolVersion)
494     }
495     if n > maxCiphertext {
496         return c.sendAlert(alertRecordOverflow)
497     }
498     if !c.haveVers {
499         // First message, be extra suspicious:
500         // this might not be a TLS client.
501         // Bail out before reading a full 'body', if
502         // The current max version is 3.1.
503         // If the version is >= 16.0, it's probably
504         // Similarly, a clientHello message encodes
505         // well under a kilobyte. If the length is
506         // it's probably not real.
507         if (typ != recordTypeAlert && typ != want) |
508             return c.sendAlert(alertUnexpectedMe
509         }
510     }
511     if err := b.readFromUntil(c.conn, recordHeaderLen+n)
512         if err == io.EOF {
513             err = io.ErrUnexpectedEOF
514         }
515         if e, ok := err.(net.Error); !ok || !e.Tempo
516             c.setError(err)
517         }
518         return err
519     }
520
521     // Process message.
522     b, c.rawInput = c.in.splitBlock(b, recordHeaderLen+n
523     b.off = recordHeaderLen
524     if ok, err := c.in.decrypt(b); !ok {
525         return c.sendAlert(err)
526     }
527     data := b.data[b.off:]
528     if len(data) > maxPlaintext {
529         c.sendAlert(alertRecordOverflow)
530         c.in.freeBlock(b)
531         return c.error()
532     }
533
534     switch typ {
535     default:
536         c.sendAlert(alertUnexpectedMessage)
537
538     case recordTypeAlert:

```

```

539         if len(data) != 2 {
540             c.sendAlert(alertUnexpectedMessage)
541             break
542         }
543         if alert(data[1]) == alertCloseNotify {
544             c.setError(io.EOF)
545             break
546         }
547         switch data[0] {
548         case alertLevelWarning:
549             // drop on the floor
550             c.in.freeBlock(b)
551             goto Again
552         case alertLevelError:
553             c.setError(&net.OpError{Op: "remote
554         default:
555             c.sendAlert(alertUnexpectedMessage)
556         }
557
558     case recordTypeChangeCipherSpec:
559         if typ != want || len(data) != 1 || data[0]
560             c.sendAlert(alertUnexpectedMessage)
561             break
562         }
563         err := c.in.changeCipherSpec()
564         if err != nil {
565             c.sendAlert(err.(alert))
566         }
567
568     case recordTypeApplicationData:
569         if typ != want {
570             c.sendAlert(alertUnexpectedMessage)
571             break
572         }
573         c.input = b
574         b = nil
575
576     case recordTypeHandshake:
577         // TODO(rsc): Should at least pick off conne
578         if typ != want {
579             return c.sendAlert(alertNoRenegotiat
580         }
581         c.hand.Write(data)
582     }
583
584     if b != nil {
585         c.in.freeBlock(b)
586     }
587     return c.error()

```

```

588 }
589
590 // sendAlert sends a TLS alert message.
591 // c.out.Mutex <= L.
592 func (c *Conn) sendAlertLocked(err alert) error {
593     c.tmp[0] = alertLevelError
594     if err == alertNoRenegotiation {
595         c.tmp[0] = alertLevelWarning
596     }
597     c.tmp[1] = byte(err)
598     c.writeRecord(recordTypeAlert, c.tmp[0:2])
599     // closeNotify is a special case in that it isn't an
600     if err != alertCloseNotify {
601         return c.setError(&net.OpError{Op: "local er
602     }
603     return nil
604 }
605
606 // sendAlert sends a TLS alert message.
607 // L < c.out.Mutex.
608 func (c *Conn) sendAlert(err alert) error {
609     c.out.Lock()
610     defer c.out.Unlock()
611     return c.sendAlertLocked(err)
612 }
613
614 // writeRecord writes a TLS record with the given type and p
615 // to the connection and updates the record layer state.
616 // c.out.Mutex <= L.
617 func (c *Conn) writeRecord(typ recordType, data []byte) (n i
618     b := c.out.newBlock()
619     for len(data) > 0 {
620         m := len(data)
621         if m > maxPlaintext {
622             m = maxPlaintext
623         }
624         b.resize(recordHeaderLen + m)
625         b.data[0] = byte(typ)
626         vers := c.vers
627         if vers == 0 {
628             vers = maxVersion
629         }
630         b.data[1] = byte(vers >> 8)
631         b.data[2] = byte(vers)
632         b.data[3] = byte(m >> 8)
633         b.data[4] = byte(m)
634         copy(b.data[recordHeaderLen:], data)
635         c.out.encrypt(b)
636         _, err = c.conn.Write(b.data)

```

```

637         if err != nil {
638             break
639         }
640         n += m
641         data = data[m:]
642     }
643     c.out.freeBlock(b)
644
645     if typ == recordTypeChangeCipherSpec {
646         err = c.out.changeCipherSpec()
647         if err != nil {
648             // Cannot call sendAlert directly,
649             // because we already hold c.out.Mut
650             c.tmp[0] = alertLevelError
651             c.tmp[1] = byte(err.(alert))
652             c.writeRecord(recordTypeAlert, c.tmp
653             c.err = &net.OpError{Op: "local erro
654             return n, c.err
655         }
656     }
657     return
658 }
659
660 // readHandshake reads the next handshake message from
661 // the record layer.
662 // c.in.Mutex < L; c.out.Mutex < L.
663 func (c *Conn) readHandshake() (interface{}, error) {
664     for c.hand.Len() < 4 {
665         if c.err != nil {
666             return nil, c.err
667         }
668         if err := c.readRecord(recordTypeHandshake);
669             return nil, err
670     }
671 }
672
673     data := c.hand.Bytes()
674     n := int(data[1])<<16 | int(data[2])<<8 | int(data[3
675     if n > maxHandshake {
676         c.sendAlert(alertInternalError)
677         return nil, c.err
678     }
679     for c.hand.Len() < 4+n {
680         if c.err != nil {
681             return nil, c.err
682         }
683         if err := c.readRecord(recordTypeHandshake);
684             return nil, err
685     }
686 }

```

```

687     data = c.hand.Next(4 + n)
688     var m handshakeMessage
689     switch data[0] {
690     case typeClientHello:
691         m = new(clientHelloMsg)
692     case typeServerHello:
693         m = new(serverHelloMsg)
694     case typeCertificate:
695         m = new(certificateMsg)
696     case typeCertificateRequest:
697         m = new(certificateRequestMsg)
698     case typeCertificateStatus:
699         m = new(certificateStatusMsg)
700     case typeServerKeyExchange:
701         m = new(serverKeyExchangeMsg)
702     case typeServerHelloDone:
703         m = new(serverHelloDoneMsg)
704     case typeClientKeyExchange:
705         m = new(clientKeyExchangeMsg)
706     case typeCertificateVerify:
707         m = new(certificateVerifyMsg)
708     case typeNextProtocol:
709         m = new(nextProtoMsg)
710     case typeFinished:
711         m = new(finishedMsg)
712     default:
713         c.sendAlert(alertUnexpectedMessage)
714         return nil, alertUnexpectedMessage
715     }
716
717     // The handshake message unmarshallers
718     // expect to be able to keep references to data,
719     // so pass in a fresh copy that won't be overwritten
720     data = append([]byte(nil), data...)
721
722     if !m.unmarshal(data) {
723         c.sendAlert(alertUnexpectedMessage)
724         return nil, alertUnexpectedMessage
725     }
726     return m, nil
727 }
728
729 // Write writes data to the connection.
730 func (c *Conn) Write(b []byte) (int, error) {
731     if c.err != nil {
732         return 0, c.err
733     }
734
735     if c.err = c.Handshake(); c.err != nil {

```

```

736         return 0, c.err
737     }
738
739     c.out.Lock()
740     defer c.out.Unlock()
741
742     if !c.handshakeComplete {
743         return 0, alertInternalError
744     }
745
746     var n int
747     n, c.err = c.writeRecord(recordTypeApplicationData,
748     return n, c.err
749 }
750
751 // Read can be made to time out and return a net.Error with
752 // after a fixed time limit; see SetDeadline and SetReadDead
753 func (c *Conn) Read(b []byte) (n int, err error) {
754     if err = c.Handshake(); err != nil {
755         return
756     }
757
758     c.in.Lock()
759     defer c.in.Unlock()
760
761     for c.input == nil && c.err == nil {
762         if err := c.readRecord(recordTypeApplication
763             // Soft error, like EAGAIN
764             return 0, err
765         }
766     }
767     if c.err != nil {
768         return 0, c.err
769     }
770     n, err = c.input.Read(b)
771     if c.input.off >= len(c.input.data) {
772         c.in.freeBlock(c.input)
773         c.input = nil
774     }
775     return n, nil
776 }
777
778 // Close closes the connection.
779 func (c *Conn) Close() error {
780     var alertErr error
781
782     c.handshakeMutex.Lock()
783     defer c.handshakeMutex.Unlock()
784     if c.handshakeComplete {

```

```

785         alertErr = c.sendAlert(alertCloseNotify)
786     }
787
788     if err := c.conn.Close(); err != nil {
789         return err
790     }
791     return alertErr
792 }
793
794 // Handshake runs the client or server handshake
795 // protocol if it has not yet been run.
796 // Most uses of this package need not call Handshake
797 // explicitly: the first Read or Write will call it automati
798 func (c *Conn) Handshake() error {
799     c.handshakeMutex.Lock()
800     defer c.handshakeMutex.Unlock()
801     if err := c.error(); err != nil {
802         return err
803     }
804     if c.handshakeComplete {
805         return nil
806     }
807     if c.isClient {
808         return c.clientHandshake()
809     }
810     return c.serverHandshake()
811 }
812
813 // ConnectionState returns basic TLS details about the conne
814 func (c *Conn) ConnectionState() ConnectionState {
815     c.handshakeMutex.Lock()
816     defer c.handshakeMutex.Unlock()
817
818     var state ConnectionState
819     state.HandshakeComplete = c.handshakeComplete
820     if c.handshakeComplete {
821         state.NegotiatedProtocol = c.clientProtocol
822         state.NegotiatedProtocolIsMutual = !c.client
823         state.CipherSuite = c.cipherSuite
824         state.PeerCertificates = c.peerCertificates
825         state.VerifiedChains = c.verifiedChains
826         state.ServerName = c.serverName
827     }
828
829     return state
830 }
831
832 // OCSPResponse returns the stapled OCSP response from the T
833 // any. (Only valid for client connections.)
834 func (c *Conn) OCSPResponse() []byte {

```

```
835         c.handshakeMutex.Lock()
836         defer c.handshakeMutex.Unlock()
837
838         return c.ocspResponse
839     }
840
841     // VerifyHostname checks that the peer certificate chain is
842     // connecting to host. If so, it returns nil; if not, it re
843     // describing the problem.
844     func (c *Conn) VerifyHostname(host string) error {
845         c.handshakeMutex.Lock()
846         defer c.handshakeMutex.Unlock()
847         if !c.isClient {
848             return errors.New("VerifyHostname called on
849         }
850         if !c.handshakeComplete {
851             return errors.New("TLS handshake has not yet
852         }
853         return c.peerCertificates[0].VerifyHostname(host)
854     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/tls/handshake_client.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tls
6
7 import (
8     "bytes"
9     "crypto"
10    "crypto/rsa"
11    "crypto/subtle"
12    "crypto/x509"
13    "errors"
14    "io"
15    "strconv"
16 )
17
18 func (c *Conn) clientHandshake() error {
19     finishedHash := newFinishedHash(versionTLS10)
20
21     if c.config == nil {
22         c.config = defaultConfig()
23     }
24
25     hello := &clientHelloMsg{
26         vers:                maxVersion,
27         cipherSuites:         c.config.cipherSuites(),
28         compressionMethods:  []uint8{compressionNone}
29         random:               make([]byte, 32),
30         ocspStapling:         true,
31         serverName:           c.config.ServerName,
32         supportedCurves:     []uint16{curveP256, curv
33         supportedPoints:    []uint8{pointFormatUncor
34         nextProtoNeg:       len(c.config.NextProtos)
35     }
36
37     t := uint32(c.config.time().Unix())
38     hello.random[0] = byte(t >> 24)
39     hello.random[1] = byte(t >> 16)
40     hello.random[2] = byte(t >> 8)
41     hello.random[3] = byte(t)
```

```

42     _, err := io.ReadFull(c.config.rand(), hello.random[
43     if err != nil {
44         c.sendAlert(alertInternalError)
45         return errors.New("short read from Rand")
46     }
47
48     finishedHash.Write(hello.marshal())
49     c.writeRecord(recordTypeHandshake, hello.marshal())
50
51     msg, err := c.readHandshake()
52     if err != nil {
53         return err
54     }
55     serverHello, ok := msg.(*serverHelloMsg)
56     if !ok {
57         return c.sendAlert(alertUnexpectedMessage)
58     }
59     finishedHash.Write(serverHello.marshal())
60
61     vers, ok := mutualVersion(serverHello.vers)
62     if !ok || vers < versionTLS10 {
63         // TLS 1.0 is the minimum version supported
64         return c.sendAlert(alertProtocolVersion)
65     }
66     c.vers = vers
67     c.haveVers = true
68
69     if serverHello.compressionMethod != compressionNone
70         return c.sendAlert(alertUnexpectedMessage)
71     }
72
73     if !hello.nextProtoNeg && serverHello.nextProtoNeg {
74         c.sendAlert(alertHandshakeFailure)
75         return errors.New("server advertised unrequ
76     }
77
78     suite := mutualCipherSuite(c.config.cipherSuites(),
79     if suite == nil {
80         return c.sendAlert(alertHandshakeFailure)
81     }
82
83     msg, err = c.readHandshake()
84     if err != nil {
85         return err
86     }
87     certMsg, ok := msg.(*certificateMsg)
88     if !ok || len(certMsg.certificates) == 0 {
89         return c.sendAlert(alertUnexpectedMessage)
90     }
91     finishedHash.Write(certMsg.marshal())

```

```

92
93     certs := make([]*x509.Certificate, len(certMsg.certi
94     for i, asn1Data := range certMsg.certificates {
95         cert, err := x509.ParseCertificate(asn1Data)
96         if err != nil {
97             c.sendAlert(alertBadCertificate)
98             return errors.New("failed to parse c
99         }
100        certs[i] = cert
101    }
102
103    if !c.config.InsecureSkipVerify {
104        opts := x509.VerifyOptions{
105            Roots:         c.config.RootCAs,
106            CurrentTime:   c.config.time(),
107            DNSName:       c.config.ServerName,
108            Intermediates: x509.NewCertPool(),
109        }
110
111        for i, cert := range certs {
112            if i == 0 {
113                continue
114            }
115            opts.Intermediates.AddCert(cert)
116        }
117        c.verifiedChains, err = certs[0].Verify(opts
118        if err != nil {
119            c.sendAlert(alertBadCertificate)
120            return err
121        }
122    }
123
124    if _, ok := certs[0].PublicKey.(*rsa.PublicKey); !ok
125        return c.sendAlert(alertUnsupportedCertifica
126    }
127
128    c.peerCertificates = certs
129
130    if serverHello.ocspStapling {
131        msg, err = c.readHandshake()
132        if err != nil {
133            return err
134        }
135        cs, ok := msg.(*certificateStatusMsg)
136        if !ok {
137            return c.sendAlert(alertUnexpectedMe
138        }
139        finishedHash.Write(cs.marshal())
140

```

```

141         if cs.statusType == statusTypeOCSP {
142             c.ocspResponse = cs.response
143         }
144     }
145
146     msg, err = c.readHandshake()
147     if err != nil {
148         return err
149     }
150
151     keyAgreement := suite.ka()
152
153     skx, ok := msg.(*serverKeyExchangeMsg)
154     if ok {
155         finishedHash.Write(skx.marshal())
156         err = keyAgreement.processServerKeyExchange(
157             if err != nil {
158                 c.sendAlert(alertUnexpectedMessage)
159                 return err
160             }
161
162             msg, err = c.readHandshake()
163             if err != nil {
164                 return err
165             }
166         }
167
168     var certToSend *Certificate
169     var certRequested bool
170     certReq, ok := msg.(*certificateRequestMsg)
171     if ok {
172         certRequested = true
173
174         // RFC 4346 on the certificateAuthorities fi
175         // A list of the distinguished names of acce
176         // authorities. These distinguished names ma
177         // distinguished name for a root CA or for a
178         // thus, this message can be used to describ
179         // and a desired authorization space. If the
180         // certificate_authorities list is empty the
181         // send any certificate of the appropriate
182         // ClientCertificateType, unless there is so
183         // arrangement to the contrary.
184
185         finishedHash.Write(certReq.marshal())
186
187         // For now, we only know how to sign challen
188         rsaAvail := false
189         for _, certType := range certReq.certificate

```

```

190         if certType == certTypeRSASign {
191             rsaAvail = true
192             break
193         }
194     }
195
196     // We need to search our list of client cert
197     // where SignatureAlgorithm is RSA and the I
198     // certReq.certificateAuthorities
199 findCert:
200     for i, cert := range c.config.Certificates {
201         if !rsaAvail {
202             continue
203         }
204
205         leaf := cert.Leaf
206         if leaf == nil {
207             if leaf, err = x509.ParseCer
208                 c.sendAlert(alertInt
209                 return errors.New("t
210             }
211         }
212
213         if leaf.PublicKeyAlgorithm != x509.R
214             continue
215         }
216
217         if len(certReq.certificateAuthoritie
218             // they gave us an empty lis
219             // first RSA cert from c.con
220             certToSend = &cert
221             break
222         }
223
224         for _, ca := range certReq.certifica
225             if bytes.Equal(leaf.RawIssue
226                 certToSend = &cert
227                 break findCert
228             }
229         }
230     }
231
232     msg, err = c.readHandshake()
233     if err != nil {
234         return err
235     }
236 }
237
238 shd, ok := msg.(*serverHelloDoneMsg)
239 if !ok {

```

```

240         return c.sendAlert(alertUnexpectedMessage)
241     }
242     finishedHash.Write(shd.marshal())
243
244     // If the server requested a certificate then we hav
245     // Certificate message, even if it's empty because w
246     // certificate to send.
247     if certRequested {
248         certMsg = new(certificateMsg)
249         if certToSend != nil {
250             certMsg.certificates = certToSend.Ce
251         }
252         finishedHash.Write(certMsg.marshal())
253         c.writeRecord(recordTypeHandshake, certMsg.m
254     }
255
256     preMasterSecret, ckx, err := keyAgreement.generateCl
257     if err != nil {
258         c.sendAlert(alertInternalError)
259         return err
260     }
261     if ckx != nil {
262         finishedHash.Write(ckx.marshal())
263         c.writeRecord(recordTypeHandshake, ckx.marsh
264     }
265
266     if certToSend != nil {
267         certVerify := new(certificateVerifyMsg)
268         digest := make([]byte, 0, 36)
269         digest = finishedHash.serverMD5.Sum(digest)
270         digest = finishedHash.serverSHA1.Sum(digest)
271         signed, err := rsa.SignPKCS1v15(c.config.rand
272         if err != nil {
273             return c.sendAlert(alertInternalErro
274         }
275         certVerify.signature = signed
276
277         finishedHash.Write(certVerify.marshal())
278         c.writeRecord(recordTypeHandshake, certVerif
279     }
280
281     masterSecret, clientMAC, serverMAC, clientKey, serve
282         keysFromPreMasterSecret(c.vers, preMasterSec
283
284     clientCipher := suite.cipher(clientKey, clientIV, fa
285     clientHash := suite.mac(c.vers, clientMAC)
286     c.out.prepareCipherSpec(c.vers, clientCipher, client
287     c.writeRecord(recordTypeChangeCipherSpec, []byte{1})
288

```

```

289     if serverHello.nextProtoNeg {
290         nextProto := new(nextProtoMsg)
291         proto, fallback := mutualProtocol(c.config.N
292         nextProto.proto = proto
293         c.clientProtocol = proto
294         c.clientProtocolFallback = fallback
295
296         finishedHash.Write(nextProto.marshal())
297         c.writeRecord(recordTypeHandshake, nextProto
298     }
299
300     finished := new(finishedMsg)
301     finished.verifyData = finishedHash.clientSum(masterS
302     finishedHash.Write(finished.marshal())
303     c.writeRecord(recordTypeHandshake, finished.marshal(
304
305     serverCipher := suite.cipher(serverKey, serverIV, tr
306     serverHash := suite.mac(c.vers, serverMAC)
307     c.in.prepareCipherSpec(c.vers, serverCipher, serverH
308     c.readRecord(recordTypeChangeCipherSpec)
309     if c.err != nil {
310         return c.err
311     }
312
313     msg, err = c.readHandshake()
314     if err != nil {
315         return err
316     }
317     serverFinished, ok := msg.(*finishedMsg)
318     if !ok {
319         return c.sendAlert(alertUnexpectedMessage)
320     }
321
322     verify := finishedHash.serverSum(masterSecret)
323     if len(verify) != len(serverFinished.verifyData) ||
324         subtle.ConstantTimeCompare(verify, serverFin
325         return c.sendAlert(alertHandshakeFailure)
326     }
327
328     c.handshakeComplete = true
329     c.cipherSuite = suite.id
330     return nil
331 }
332
333 // mutualProtocol finds the mutual Next Protocol Negotiation
334 // set of client and server supported protocols. The set of
335 // protocols must not be empty. It returns the resulting pro
336 // indicating if the fallback case was reached.
337 func mutualProtocol(clientProtos, serverProtos []string) (st

```

```
338     for _, s := range serverProtos {
339         for _, c := range clientProtos {
340             if s == c {
341                 return s, false
342             }
343         }
344     }
345     return clientProtos[0], true
346 }
347 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/tls/handshake_message

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tls
6
7 import "bytes"
8
9 type clientHelloMsg struct {
10     raw          []byte
11     vers         uint16
12     random       []byte
13     sessionId    []byte
14     cipherSuites []uint16
15     compressionMethods []uint8
16     nextProtoNeg  bool
17     serverName    string
18     ocsStapling  bool
19     supportedCurves []uint16
20     supportedPoints []uint8
21 }
22
23 func (m *clientHelloMsg) equal(i interface{}) bool {
24     m1, ok := i.(*clientHelloMsg)
25     if !ok {
26         return false
27     }
28
29     return bytes.Equal(m.raw, m1.raw) &&
30         m.vers == m1.vers &&
31         bytes.Equal(m.random, m1.random) &&
32         bytes.Equal(m.sessionId, m1.sessionId) &&
33         eqUint16s(m.cipherSuites, m1.cipherSuites) &
34         bytes.Equal(m.compressionMethods, m1.compressionMethods) &&
35         m.nextProtoNeg == m1.nextProtoNeg &&
36         m.serverName == m1.serverName &&
37         m.ocsStapling == m1.ocsStapling &&
38         eqUint16s(m.supportedCurves, m1.supportedCurves) &&
39         bytes.Equal(m.supportedPoints, m1.supportedPoints)
40 }
41
```

```

42 func (m *clientHelloMsg) marshal() []byte {
43     if m.raw != nil {
44         return m.raw
45     }
46
47     length := 2 + 32 + 1 + len(m.sessionId) + 2 + len(m.
48     numExtensions := 0
49     extensionsLength := 0
50     if m.nextProtoNeg {
51         numExtensions++
52     }
53     if m.ocspStapling {
54         extensionsLength += 1 + 2 + 2
55         numExtensions++
56     }
57     if len(m.serverName) > 0 {
58         extensionsLength += 5 + len(m.serverName)
59         numExtensions++
60     }
61     if len(m.supportedCurves) > 0 {
62         extensionsLength += 2 + 2*len(m.supportedCur
63         numExtensions++
64     }
65     if len(m.supportedPoints) > 0 {
66         extensionsLength += 1 + len(m.supportedPoint
67         numExtensions++
68     }
69     if numExtensions > 0 {
70         extensionsLength += 4 * numExtensions
71         length += 2 + extensionsLength
72     }
73
74     x := make([]byte, 4+length)
75     x[0] = typeClientHello
76     x[1] = uint8(length >> 16)
77     x[2] = uint8(length >> 8)
78     x[3] = uint8(length)
79     x[4] = uint8(m.vers >> 8)
80     x[5] = uint8(m.vers)
81     copy(x[6:38], m.random)
82     x[38] = uint8(len(m.sessionId))
83     copy(x[39:39+len(m.sessionId)], m.sessionId)
84     y := x[39+len(m.sessionId):]
85     y[0] = uint8(len(m.cipherSuites) >> 7)
86     y[1] = uint8(len(m.cipherSuites) << 1)
87     for i, suite := range m.cipherSuites {
88         y[2+i*2] = uint8(suite >> 8)
89         y[3+i*2] = uint8(suite)
90     }
91     z := y[2+len(m.cipherSuites)*2:]

```

```

92     z[0] = uint8(len(m.compressionMethods))
93     copy(z[1:], m.compressionMethods)
94
95     z = z[1+len(m.compressionMethods):]
96     if numExtensions > 0 {
97         z[0] = byte(extensionsLength >> 8)
98         z[1] = byte(extensionsLength)
99         z = z[2:]
100    }
101    if m.nextProtoNeg {
102        z[0] = byte(extensionNextProtoNeg >> 8)
103        z[1] = byte(extensionNextProtoNeg)
104        // The length is always 0
105        z = z[4:]
106    }
107    if len(m.serverName) > 0 {
108        z[0] = byte(extensionServerName >> 8)
109        z[1] = byte(extensionServerName)
110        l := len(m.serverName) + 5
111        z[2] = byte(l >> 8)
112        z[3] = byte(l)
113        z = z[4:]
114
115        // RFC 3546, section 3.1
116        //
117        // struct {
118        //     NameType name_type;
119        //     select (name_type) {
120        //         case host_name: HostName;
121        //     } name;
122        // } ServerName;
123        //
124        // enum {
125        //     host_name(0), (255)
126        // } NameType;
127        //
128        // opaque HostName<1..2^16-1>;
129        //
130        // struct {
131        //     ServerName server_name_list<1..2^16-1
132        // } ServerNameList;
133
134        z[0] = byte((len(m.serverName) + 3) >> 8)
135        z[1] = byte(len(m.serverName) + 3)
136        z[3] = byte(len(m.serverName) >> 8)
137        z[4] = byte(len(m.serverName))
138        copy(z[5:], []byte(m.serverName))
139        z = z[1:]
140    }

```

```

141     if m.ocspStapling {
142         // RFC 4366, section 3.6
143         z[0] = byte(extensionStatusRequest >> 8)
144         z[1] = byte(extensionStatusRequest)
145         z[2] = 0
146         z[3] = 5
147         z[4] = 1 // OCSP type
148         // Two zero valued uint16s for the two lengt
149         z = z[9:]
150     }
151     if len(m.supportedCurves) > 0 {
152         // http://tools.ietf.org/html/rfc4492#sectio
153         z[0] = byte(extensionSupportedCurves >> 8)
154         z[1] = byte(extensionSupportedCurves)
155         l := 2 + 2*len(m.supportedCurves)
156         z[2] = byte(l >> 8)
157         z[3] = byte(l)
158         l -= 2
159         z[4] = byte(l >> 8)
160         z[5] = byte(l)
161         z = z[6:]
162         for _, curve := range m.supportedCurves {
163             z[0] = byte(curve >> 8)
164             z[1] = byte(curve)
165             z = z[2:]
166         }
167     }
168     if len(m.supportedPoints) > 0 {
169         // http://tools.ietf.org/html/rfc4492#sectio
170         z[0] = byte(extensionSupportedPoints >> 8)
171         z[1] = byte(extensionSupportedPoints)
172         l := 1 + len(m.supportedPoints)
173         z[2] = byte(l >> 8)
174         z[3] = byte(l)
175         l--
176         z[4] = byte(l)
177         z = z[5:]
178         for _, pointFormat := range m.supportedPoint
179             z[0] = byte(pointFormat)
180             z = z[1:]
181         }
182     }
183     m.raw = x
184     return x
185 }
186 }
187 }
188 }
189 func (m *clientHelloMsg) unmarshal(data []byte) bool {

```

```

190     if len(data) < 42 {
191         return false
192     }
193     m.raw = data
194     m.vers = uint16(data[4])<<8 | uint16(data[5])
195     m.random = data[6:38]
196     sessionIdLen := int(data[38])
197     if sessionIdLen > 32 || len(data) < 39+sessionIdLen
198         return false
199     }
200     m.sessionId = data[39 : 39+sessionIdLen]
201     data = data[39+sessionIdLen:]
202     if len(data) < 2 {
203         return false
204     }
205     // cipherSuiteLen is the number of bytes of cipher s
206     // they are uint16s, the number must be even.
207     cipherSuiteLen := int(data[0])<<8 | int(data[1])
208     if cipherSuiteLen%2 == 1 || len(data) < 2+cipherSuit
209         return false
210     }
211     numCipherSuites := cipherSuiteLen / 2
212     m.cipherSuites = make([]uint16, numCipherSuites)
213     for i := 0; i < numCipherSuites; i++ {
214         m.cipherSuites[i] = uint16(data[2+2*i])<<8 |
215     }
216     data = data[2+cipherSuiteLen:]
217     if len(data) < 1 {
218         return false
219     }
220     compressionMethodsLen := int(data[0])
221     if len(data) < 1+compressionMethodsLen {
222         return false
223     }
224     m.compressionMethods = data[1 : 1+compressionMethods
225
226     data = data[1+compressionMethodsLen:]
227
228     m.nextProtoNeg = false
229     m.serverName = ""
230     m.ocspStapling = false
231
232     if len(data) == 0 {
233         // ClientHello is optionally followed by ext
234         return true
235     }
236     if len(data) < 2 {
237         return false
238     }
239

```

```

240     extensionsLength := int(data[0])<<8 | int(data[1])
241     data = data[2:]
242     if extensionsLength != len(data) {
243         return false
244     }
245
246     for len(data) != 0 {
247         if len(data) < 4 {
248             return false
249         }
250         extension := uint16(data[0])<<8 | uint16(dat
251         length := int(data[2])<<8 | int(data[3])
252         data = data[4:]
253         if len(data) < length {
254             return false
255         }
256
257         switch extension {
258         case extensionServerName:
259             if length < 2 {
260                 return false
261             }
262             numNames := int(data[0])<<8 | int(da
263             d := data[2:]
264             for i := 0; i < numNames; i++ {
265                 if len(d) < 3 {
266                     return false
267                 }
268                 nameType := d[0]
269                 nameLen := int(d[1])<<8 | in
270                 d = d[3:]
271                 if len(d) < nameLen {
272                     return false
273                 }
274                 if nameType == 0 {
275                     m.serverName = strin
276                     break
277                 }
278                 d = d[nameLen:]
279             }
280         case extensionNextProtoNeg:
281             if length > 0 {
282                 return false
283             }
284             m.nextProtoNeg = true
285         case extensionStatusRequest:
286             m.ocspStapling = length > 0 && data[
287         case extensionSupportedCurves:
288             // http://tools.ietf.org/html/rfc449

```

```

289         if length < 2 {
290             return false
291         }
292         l := int(data[0])<<8 | int(data[1])
293         if l%2 == 1 || length != l+2 {
294             return false
295         }
296         numCurves := l / 2
297         m.supportedCurves = make([]uint16, n
298         d := data[2:]
299         for i := 0; i < numCurves; i++ {
300             m.supportedCurves[i] = uint1
301             d = d[2:]
302         }
303         case extensionSupportedPoints:
304             // http://tools.ietf.org/html/rfc449
305             if length < 1 {
306                 return false
307             }
308             l := int(data[0])
309             if length != l+1 {
310                 return false
311             }
312             m.supportedPoints = make([]uint8, l)
313             copy(m.supportedPoints, data[1:])
314         }
315         data = data[length:]
316     }
317
318     return true
319 }
320
321 type serverHelloMsg struct {
322     raw          []byte
323     vers         uint16
324     random       []byte
325     sessionId    []byte
326     cipherSuite  uint16
327     compressionMethod uint8
328     nextProtoNeg bool
329     nextProtos   []string
330     ocsStapling  bool
331 }
332
333 func (m *serverHelloMsg) equal(i interface{}) bool {
334     m1, ok := i.(*serverHelloMsg)
335     if !ok {
336         return false
337     }

```

```

338
339     return bytes.Equal(m.raw, m1.raw) &&
340         m.vers == m1.vers &&
341         bytes.Equal(m.random, m1.random) &&
342         bytes.Equal(m.sessionId, m1.sessionId) &&
343         m.cipherSuite == m1.cipherSuite &&
344         m.compressionMethod == m1.compressionMethod
345         m.nextProtoNeg == m1.nextProtoNeg &&
346         eqStrings(m.nextProtos, m1.nextProtos) &&
347         m.ocspStapling == m1.ocspStapling
348 }
349
350 func (m *serverHelloMsg) marshal() []byte {
351     if m.raw != nil {
352         return m.raw
353     }
354
355     length := 38 + len(m.sessionId)
356     numExtensions := 0
357     extensionsLength := 0
358
359     nextProtoLen := 0
360     if m.nextProtoNeg {
361         numExtensions++
362         for _, v := range m.nextProtos {
363             nextProtoLen += len(v)
364         }
365         nextProtoLen += len(m.nextProtos)
366         extensionsLength += nextProtoLen
367     }
368     if m.ocspStapling {
369         numExtensions++
370     }
371     if numExtensions > 0 {
372         extensionsLength += 4 * numExtensions
373         length += 2 + extensionsLength
374     }
375
376     x := make([]byte, 4+length)
377     x[0] = typeServerHello
378     x[1] = uint8(length >> 16)
379     x[2] = uint8(length >> 8)
380     x[3] = uint8(length)
381     x[4] = uint8(m.vers >> 8)
382     x[5] = uint8(m.vers)
383     copy(x[6:38], m.random)
384     x[38] = uint8(len(m.sessionId))
385     copy(x[39:39+len(m.sessionId)], m.sessionId)
386     z := x[39+len(m.sessionId):]
387     z[0] = uint8(m.cipherSuite >> 8)

```

```

388     z[1] = uint8(m.cipherSuite)
389     z[2] = uint8(m.compressionMethod)
390
391     z = z[3:]
392     if numExtensions > 0 {
393         z[0] = byte(extensionsLength >> 8)
394         z[1] = byte(extensionsLength)
395         z = z[2:]
396     }
397     if m.nextProtoNeg {
398         z[0] = byte(extensionNextProtoNeg >> 8)
399         z[1] = byte(extensionNextProtoNeg)
400         z[2] = byte(nextProtoLen >> 8)
401         z[3] = byte(nextProtoLen)
402         z = z[4:]
403
404         for _, v := range m.nextProtos {
405             l := len(v)
406             if l > 255 {
407                 l = 255
408             }
409             z[0] = byte(l)
410             copy(z[1:], []byte(v[0:l]))
411             z = z[1+l:]
412         }
413     }
414     if m.ocspStapling {
415         z[0] = byte(extensionStatusRequest >> 8)
416         z[1] = byte(extensionStatusRequest)
417         z = z[4:]
418     }
419
420     m.raw = x
421
422     return x
423 }
424
425 func (m *serverHelloMsg) unmarshal(data []byte) bool {
426     if len(data) < 42 {
427         return false
428     }
429     m.raw = data
430     m.vers = uint16(data[4])<<8 | uint16(data[5])
431     m.random = data[6:38]
432     sessionIdLen := int(data[38])
433     if sessionIdLen > 32 || len(data) < 39+sessionIdLen
434         return false
435     }
436     m.sessionId = data[39 : 39+sessionIdLen]

```

```

437     data = data[39+sessionIdLen:]
438     if len(data) < 3 {
439         return false
440     }
441     m.cipherSuite = uint16(data[0])<<8 | uint16(data[1])
442     m.compressionMethod = data[2]
443     data = data[3:]
444
445     m.nextProtoNeg = false
446     m.nextProtos = nil
447     m.ocspStapling = false
448
449     if len(data) == 0 {
450         // ServerHello is optionally followed by ext
451         return true
452     }
453     if len(data) < 2 {
454         return false
455     }
456
457     extensionsLength := int(data[0])<<8 | int(data[1])
458     data = data[2:]
459     if len(data) != extensionsLength {
460         return false
461     }
462
463     for len(data) != 0 {
464         if len(data) < 4 {
465             return false
466         }
467         extension := uint16(data[0])<<8 | uint16(data[1])
468         length := int(data[2])<<8 | int(data[3])
469         data = data[4:]
470         if len(data) < length {
471             return false
472         }
473
474         switch extension {
475         case extensionNextProtoNeg:
476             m.nextProtoNeg = true
477             d := data
478             for len(d) > 0 {
479                 l := int(d[0])
480                 d = d[1:]
481                 if l == 0 || l > len(d) {
482                     return false
483                 }
484                 m.nextProtos = append(m.nextProtos, d[:l])
485                 d = d[l:]

```

```

486         }
487         case extensionStatusRequest:
488             if length > 0 {
489                 return false
490             }
491             m.ocspStapling = true
492         }
493         data = data[length:]
494     }
495
496     return true
497 }
498
499 type certificateMsg struct {
500     raw []byte
501     certificates [][]byte
502 }
503
504 func (m *certificateMsg) equal(i interface{}) bool {
505     m1, ok := i.(*certificateMsg)
506     if !ok {
507         return false
508     }
509
510     return bytes.Equal(m.raw, m1.raw) &&
511         eqByteSlices(m.certificates, m1.certificates)
512 }
513
514 func (m *certificateMsg) marshal() (x []byte) {
515     if m.raw != nil {
516         return m.raw
517     }
518
519     var i int
520     for _, slice := range m.certificates {
521         i += len(slice)
522     }
523
524     length := 3 + 3*len(m.certificates) + i
525     x = make([]byte, 4+length)
526     x[0] = typeCertificate
527     x[1] = uint8(length >> 16)
528     x[2] = uint8(length >> 8)
529     x[3] = uint8(length)
530
531     certificateOctets := length - 3
532     x[4] = uint8(certificateOctets >> 16)
533     x[5] = uint8(certificateOctets >> 8)
534     x[6] = uint8(certificateOctets)
535

```

```

536     y := x[7:]
537     for _, slice := range m.certificates {
538         y[0] = uint8(len(slice) >> 16)
539         y[1] = uint8(len(slice) >> 8)
540         y[2] = uint8(len(slice))
541         copy(y[3:], slice)
542         y = y[3+len(slice):]
543     }
544
545     m.raw = x
546     return
547 }
548
549 func (m *certificateMsg) unmarshal(data []byte) bool {
550     if len(data) < 7 {
551         return false
552     }
553
554     m.raw = data
555     certsLen := uint32(data[4])<<16 | uint32(data[5])<<8
556     if uint32(len(data)) != certsLen+7 {
557         return false
558     }
559
560     numCerts := 0
561     d := data[7:]
562     for certsLen > 0 {
563         if len(d) < 4 {
564             return false
565         }
566         certLen := uint32(d[0])<<24 | uint32(d[1])<<8
567         if uint32(len(d)) < 3+certLen {
568             return false
569         }
570         d = d[3+certLen:]
571         certsLen -= 3 + certLen
572         numCerts++
573     }
574
575     m.certificates = make([][]byte, numCerts)
576     d = data[7:]
577     for i := 0; i < numCerts; i++ {
578         certLen := uint32(d[0])<<24 | uint32(d[1])<<8
579         m.certificates[i] = d[3 : 3+certLen]
580         d = d[3+certLen:]
581     }
582
583     return true
584 }

```

```

585
586 type serverKeyExchangeMsg struct {
587     raw []byte
588     key []byte
589 }
590
591 func (m *serverKeyExchangeMsg) equal(i interface{}) bool {
592     m1, ok := i.(*serverKeyExchangeMsg)
593     if !ok {
594         return false
595     }
596
597     return bytes.Equal(m.raw, m1.raw) &&
598         bytes.Equal(m.key, m1.key)
599 }
600
601 func (m *serverKeyExchangeMsg) marshal() []byte {
602     if m.raw != nil {
603         return m.raw
604     }
605     length := len(m.key)
606     x := make([]byte, length+4)
607     x[0] = typeServerKeyExchange
608     x[1] = uint8(length >> 16)
609     x[2] = uint8(length >> 8)
610     x[3] = uint8(length)
611     copy(x[4:], m.key)
612
613     m.raw = x
614     return x
615 }
616
617 func (m *serverKeyExchangeMsg) unmarshal(data []byte) bool {
618     m.raw = data
619     if len(data) < 4 {
620         return false
621     }
622     m.key = data[4:]
623     return true
624 }
625
626 type certificateStatusMsg struct {
627     raw          []byte
628     statusType  uint8
629     response    []byte
630 }
631
632 func (m *certificateStatusMsg) equal(i interface{}) bool {
633     m1, ok := i.(*certificateStatusMsg)

```

```

634         if !ok {
635             return false
636         }
637
638         return bytes.Equal(m.raw, m1.raw) &&
639             m.statusType == m1.statusType &&
640             bytes.Equal(m.response, m1.response)
641     }
642
643     func (m *certificateStatusMsg) marshal() []byte {
644         if m.raw != nil {
645             return m.raw
646         }
647
648         var x []byte
649         if m.statusType == statusTypeOCSP {
650             x = make([]byte, 4+4+len(m.response))
651             x[0] = typeCertificateStatus
652             l := len(m.response) + 4
653             x[1] = byte(l >> 16)
654             x[2] = byte(l >> 8)
655             x[3] = byte(l)
656             x[4] = statusTypeOCSP
657
658             l -= 4
659             x[5] = byte(l >> 16)
660             x[6] = byte(l >> 8)
661             x[7] = byte(l)
662             copy(x[8:], m.response)
663         } else {
664             x = []byte{typeCertificateStatus, 0, 0, 1, m
665         }
666
667         m.raw = x
668         return x
669     }
670
671     func (m *certificateStatusMsg) unmarshal(data []byte) bool {
672         m.raw = data
673         if len(data) < 5 {
674             return false
675         }
676         m.statusType = data[4]
677
678         m.response = nil
679         if m.statusType == statusTypeOCSP {
680             if len(data) < 8 {
681                 return false
682             }
683             respLen := uint32(data[5])<<16 | uint32(data

```

```

684             if uint32(len(data)) != 4+4+respLen {
685                 return false
686             }
687             m.response = data[8:]
688         }
689         return true
690     }
691
692     type serverHelloDoneMsg struct{}
693
694     func (m *serverHelloDoneMsg) equal(i interface{}) bool {
695         _, ok := i.(*serverHelloDoneMsg)
696         return ok
697     }
698
699     func (m *serverHelloDoneMsg) marshal() []byte {
700         x := make([]byte, 4)
701         x[0] = typeServerHelloDone
702         return x
703     }
704
705     func (m *serverHelloDoneMsg) unmarshal(data []byte) bool {
706         return len(data) == 4
707     }
708
709     type clientKeyExchangeMsg struct {
710         raw          []byte
711         ciphertext    []byte
712     }
713
714     func (m *clientKeyExchangeMsg) equal(i interface{}) bool {
715         m1, ok := i.(*clientKeyExchangeMsg)
716         if !ok {
717             return false
718         }
719
720         return bytes.Equal(m.raw, m1.raw) &&
721             bytes.Equal(m.ciphertext, m1.ciphertext)
722     }
723
724     func (m *clientKeyExchangeMsg) marshal() []byte {
725         if m.raw != nil {
726             return m.raw
727         }
728         length := len(m.ciphertext)
729         x := make([]byte, length+4)
730         x[0] = typeClientKeyExchange
731         x[1] = uint8(length >> 16)
732         x[2] = uint8(length >> 8)

```

```

733         x[3] = uint8(length)
734         copy(x[4:], m.ciphertext)
735
736         m.raw = x
737         return x
738     }
739
740     func (m *clientKeyExchangeMsg) unmarshal(data []byte) bool {
741         m.raw = data
742         if len(data) < 4 {
743             return false
744         }
745         l := int(data[1])<<16 | int(data[2])<<8 | int(data[3])
746         if l != len(data)-4 {
747             return false
748         }
749         m.ciphertext = data[4:]
750         return true
751     }
752
753     type finishedMsg struct {
754         raw          []byte
755         verifyData []byte
756     }
757
758     func (m *finishedMsg) equal(i interface{}) bool {
759         m1, ok := i.(*finishedMsg)
760         if !ok {
761             return false
762         }
763
764         return bytes.Equal(m.raw, m1.raw) &&
765             bytes.Equal(m.verifyData, m1.verifyData)
766     }
767
768     func (m *finishedMsg) marshal() (x []byte) {
769         if m.raw != nil {
770             return m.raw
771         }
772
773         x = make([]byte, 4+len(m.verifyData))
774         x[0] = typeFinished
775         x[3] = byte(len(m.verifyData))
776         copy(x[4:], m.verifyData)
777         m.raw = x
778         return
779     }
780
781     func (m *finishedMsg) unmarshal(data []byte) bool {

```

```

782         m.raw = data
783         if len(data) < 4 {
784             return false
785         }
786         m.verifyData = data[4:]
787         return true
788     }
789
790     type nextProtoMsg struct {
791         raw []byte
792         proto string
793     }
794
795     func (m *nextProtoMsg) equal(i interface{}) bool {
796         m1, ok := i.(*nextProtoMsg)
797         if !ok {
798             return false
799         }
800
801         return bytes.Equal(m.raw, m1.raw) &&
802             m.proto == m1.proto
803     }
804
805     func (m *nextProtoMsg) marshal() []byte {
806         if m.raw != nil {
807             return m.raw
808         }
809         l := len(m.proto)
810         if l > 255 {
811             l = 255
812         }
813
814         padding := 32 - (l+2)%32
815         length := l + padding + 2
816         x := make([]byte, length+4)
817         x[0] = typeNextProtocol
818         x[1] = uint8(length >> 16)
819         x[2] = uint8(length >> 8)
820         x[3] = uint8(length)
821
822         y := x[4:]
823         y[0] = byte(l)
824         copy(y[1:], []byte(m.proto[0:l]))
825         y = y[1+l:]
826         y[0] = byte(padding)
827
828         m.raw = x
829
830         return x
831     }

```

```

832
833 func (m *nextProtoMsg) unmarshal(data []byte) bool {
834     m.raw = data
835
836     if len(data) < 5 {
837         return false
838     }
839     data = data[4:]
840     protoLen := int(data[0])
841     data = data[1:]
842     if len(data) < protoLen {
843         return false
844     }
845     m.proto = string(data[0:protoLen])
846     data = data[protoLen:]
847
848     if len(data) < 1 {
849         return false
850     }
851     paddingLen := int(data[0])
852     data = data[1:]
853     if len(data) != paddingLen {
854         return false
855     }
856
857     return true
858 }
859
860 type certificateRequestMsg struct {
861     raw []byte
862     certificateTypes []byte
863     certificateAuthorities [][]byte
864 }
865
866 func (m *certificateRequestMsg) equal(i interface{}) bool {
867     m1, ok := i.(*certificateRequestMsg)
868     if !ok {
869         return false
870     }
871
872     return bytes.Equal(m.raw, m1.raw) &&
873         bytes.Equal(m.certificateTypes, m1.certificateTypes) &&
874         eqByteSlices(m.certificateAuthorities, m1.certificateAuthorities)
875 }
876
877 func (m *certificateRequestMsg) marshal() (x []byte) {
878     if m.raw != nil {
879         return m.raw
880     }

```

```

881
882 // See http://tools.ietf.org/html/rfc4346#section-7.
883 length := 1 + len(m.certificateTypes) + 2
884 casLength := 0
885 for _, ca := range m.certificateAuthorities {
886     casLength += 2 + len(ca)
887 }
888 length += casLength
889
890 x = make([]byte, 4+length)
891 x[0] = typeCertificateRequest
892 x[1] = uint8(length >> 16)
893 x[2] = uint8(length >> 8)
894 x[3] = uint8(length)
895
896 x[4] = uint8(len(m.certificateTypes))
897
898 copy(x[5:], m.certificateTypes)
899 y := x[5+len(m.certificateTypes):]
900 y[0] = uint8(casLength >> 8)
901 y[1] = uint8(casLength)
902 y = y[2:]
903 for _, ca := range m.certificateAuthorities {
904     y[0] = uint8(len(ca) >> 8)
905     y[1] = uint8(len(ca))
906     y = y[2:]
907     copy(y, ca)
908     y = y[len(ca):]
909 }
910
911 m.raw = x
912 return
913 }
914
915 func (m *certificateRequestMsg) unmarshal(data []byte) bool
916     m.raw = data
917
918     if len(data) < 5 {
919         return false
920     }
921
922     length := uint32(data[1])<<16 | uint32(data[2])<<8 |
923     if uint32(len(data))-4 != length {
924         return false
925     }
926
927     numCertTypes := int(data[4])
928     data = data[5:]
929     if numCertTypes == 0 || len(data) <= numCertTypes {

```

```

930         return false
931     }
932
933     m.certificateTypes = make([]byte, numCertTypes)
934     if copy(m.certificateTypes, data) != numCertTypes {
935         return false
936     }
937
938     data = data[numCertTypes:]
939
940     if len(data) < 2 {
941         return false
942     }
943     casLength := uint16(data[0])<<8 | uint16(data[1])
944     data = data[2:]
945     if len(data) < int(casLength) {
946         return false
947     }
948     cas := make([]byte, casLength)
949     copy(cas, data)
950     data = data[casLength:]
951
952     m.certificateAuthorities = nil
953     for len(cas) > 0 {
954         if len(cas) < 2 {
955             return false
956         }
957         caLen := uint16(cas[0])<<8 | uint16(cas[1])
958         cas = cas[2:]
959
960         if len(cas) < int(caLen) {
961             return false
962         }
963
964         m.certificateAuthorities = append(m.certific
965         cas = cas[caLen:]
966     }
967     if len(data) > 0 {
968         return false
969     }
970
971     return true
972 }
973
974 type certificateVerifyMsg struct {
975     raw          []byte
976     signature []byte
977 }
978
979 func (m *certificateVerifyMsg) equal(i interface{}) bool {

```

```

980         m1, ok := i.(*certificateVerifyMsg)
981         if !ok {
982             return false
983         }
984
985         return bytes.Equal(m.raw, m1.raw) &&
986             bytes.Equal(m.signature, m1.signature)
987     }
988
989     func (m *certificateVerifyMsg) marshal() (x []byte) {
990         if m.raw != nil {
991             return m.raw
992         }
993
994         // See http://tools.ietf.org/html/rfc4346#section-7.
995         siglength := len(m.signature)
996         length := 2 + siglength
997         x = make([]byte, 4+length)
998         x[0] = typeCertificateVerify
999         x[1] = uint8(length >> 16)
1000        x[2] = uint8(length >> 8)
1001        x[3] = uint8(length)
1002        x[4] = uint8(siglength >> 8)
1003        x[5] = uint8(siglength)
1004        copy(x[6:], m.signature)
1005
1006        m.raw = x
1007
1008        return
1009    }
1010
1011    func (m *certificateVerifyMsg) unmarshal(data []byte) bool {
1012        m.raw = data
1013
1014        if len(data) < 6 {
1015            return false
1016        }
1017
1018        length := uint32(data[1])<<16 | uint32(data[2])<<8 |
1019        if uint32(len(data))-4 != length {
1020            return false
1021        }
1022
1023        siglength := int(data[4])<<8 + int(data[5])
1024        if len(data)-6 != siglength {
1025            return false
1026        }
1027
1028        m.signature = data[6:]

```

```

1029
1030     return true
1031 }
1032
1033 func eqUint16s(x, y []uint16) bool {
1034     if len(x) != len(y) {
1035         return false
1036     }
1037     for i, v := range x {
1038         if y[i] != v {
1039             return false
1040         }
1041     }
1042     return true
1043 }
1044
1045 func eqStrings(x, y []string) bool {
1046     if len(x) != len(y) {
1047         return false
1048     }
1049     for i, v := range x {
1050         if y[i] != v {
1051             return false
1052         }
1053     }
1054     return true
1055 }
1056
1057 func eqByteSlices(x, y [][]byte) bool {
1058     if len(x) != len(y) {
1059         return false
1060     }
1061     for i, v := range x {
1062         if !bytes.Equal(v, y[i]) {
1063             return false
1064         }
1065     }
1066     return true
1067 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/tls/handshake_server.g

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tls
6
7 import (
8     "crypto"
9     "crypto/rsa"
10    "crypto/subtle"
11    "crypto/x509"
12    "errors"
13    "io"
14 )
15
16 func (c *Conn) serverHandshake() error {
17     config := c.config
18     msg, err := c.readHandshake()
19     if err != nil {
20         return err
21     }
22     clientHello, ok := msg.(*clientHelloMsg)
23     if !ok {
24         return c.sendAlert(alertUnexpectedMessage)
25     }
26     vers, ok := mutualVersion(clientHello.vers)
27     if !ok {
28         return c.sendAlert(alertProtocolVersion)
29     }
30     c.vers = vers
31     c.haveVers = true
32
33     finishedHash := newFinishedHash(vers)
34     finishedHash.Write(clientHello.marshal())
35
36     hello := new(serverHelloMsg)
37
38     supportedCurve := false
39     Curves:
40     for _, curve := range clientHello.supportedCurves {
41         switch curve {
```

```

42         case curveP256, curveP384, curveP521:
43             supportedCurve = true
44             break Curves
45         }
46     }
47
48     supportedPointFormat := false
49     for _, pointFormat := range clientHello.supportedPoi
50         if pointFormat == pointFormatUncompressed {
51             supportedPointFormat = true
52             break
53         }
54     }
55
56     ellipticOk := supportedCurve && supportedPointFormat
57
58     var suite *cipherSuite
59 FindCipherSuite:
60     for _, id := range clientHello.cipherSuites {
61         for _, supported := range config.cipherSuite
62             if id == supported {
63                 var candidate *cipherSuite
64
65                 for _, s := range cipherSuit
66                     if s.id == id {
67                         candidate =
68                             break
69                     }
70                 }
71                 if candidate == nil {
72                     continue
73                 }
74                 // Don't select a ciphersuit
75                 // support for this client.
76                 if candidate.elliptic && !el
77                     continue
78                 }
79                 suite = candidate
80                 break FindCipherSuite
81             }
82         }
83     }
84
85     foundCompression := false
86     // We only support null compression, so check that t
87     for _, compression := range clientHello.compressionM
88         if compression == compressionNone {
89             foundCompression = true
90             break
91         }

```

```

92     }
93
94     if suite == nil || !foundCompression {
95         return c.sendAlert(alertHandshakeFailure)
96     }
97
98     hello.vers = vers
99     hello.cipherSuite = suite.id
100    t := uint32(config.time().Unix())
101    hello.random = make([]byte, 32)
102    hello.random[0] = byte(t >> 24)
103    hello.random[1] = byte(t >> 16)
104    hello.random[2] = byte(t >> 8)
105    hello.random[3] = byte(t)
106    _, err = io.ReadFull(config.rand(), hello.random[4:])
107    if err != nil {
108        return c.sendAlert(alertInternalError)
109    }
110    hello.compressionMethod = compressionNone
111    if clientHello.nextProtoNeg {
112        hello.nextProtoNeg = true
113        hello.nextProtos = config.NextProtos
114    }
115
116    if len(config.Certificates) == 0 {
117        return c.sendAlert(alertInternalError)
118    }
119    cert := &config.Certificates[0]
120    if len(clientHello.serverName) > 0 {
121        c.serverName = clientHello.serverName
122        cert = config.getCertificateForName(clientHe
123    }
124
125    if clientHello.ocspStapling && len(cert.OCSPStaple)
126        hello.ocspStapling = true
127    }
128
129    finishedHash.Write(hello.marshal())
130    c.writeRecord(recordTypeHandshake, hello.marshal())
131
132    certMsg := new(certificateMsg)
133    certMsg.certificates = cert.Certificate
134    finishedHash.Write(certMsg.marshal())
135    c.writeRecord(recordTypeHandshake, certMsg.marshal())
136
137    if hello.ocspStapling {
138        certStatus := new(certificateStatusMsg)
139        certStatus.statusType = statusTypeOCSP
140        certStatus.response = cert.OCSPStaple

```

```

141         finishedHash.Write(certStatus.marshal())
142         c.writeRecord(recordTypeHandshake, certStatu
143     }
144
145     keyAgreement := suite.ka()
146     skx, err := keyAgreement.generateServerKeyExchange(c
147     if err != nil {
148         c.sendAlert(alertHandshakeFailure)
149         return err
150     }
151     if skx != nil {
152         finishedHash.Write(skx.marshal())
153         c.writeRecord(recordTypeHandshake, skx.marsh
154     }
155
156     if config.ClientAuth >= RequestClientCert {
157         // Request a client certificate
158         certReq := new(certificateRequestMsg)
159         certReq.certificateTypes = []byte{certTypeRS
160
161         // An empty list of certificateAuthorities s
162         // the client that it may send any certifica
163         // to our request. When we know the CAs we t
164         // we can send them down, so that the client
165         // an appropriate certificate to give to us.
166         if config.ClientCAs != nil {
167             certReq.certificateAuthorities = con
168         }
169         finishedHash.Write(certReq.marshal())
170         c.writeRecord(recordTypeHandshake, certReq.m
171     }
172
173     helloDone := new(serverHelloDoneMsg)
174     finishedHash.Write(helloDone.marshal())
175     c.writeRecord(recordTypeHandshake, helloDone.marshal
176
177     var pub *rsa.PublicKey // public key for client auth
178
179     msg, err = c.readHandshake()
180     if err != nil {
181         return err
182     }
183
184     // If we requested a client certificate, then the cl
185     // certificate message, even if it's empty.
186     if config.ClientAuth >= RequestClientCert {
187         if certMsg, ok = msg.(*certificateMsg); !ok
188             return c.sendAlert(alertHandshakeFai
189     }

```

```

190         finishedHash.Write(certMsg.marshal())
191
192     if len(certMsg.certificates) == 0 {
193         // The client didn't actually send a
194         switch config.ClientAuth {
195             case RequireAnyClientCert, RequireAn
196                 c.sendAlert(alertBadCertific
197                 return errors.New("tls: clie
198         }
199     }
200
201     certs := make([]*x509.Certificate, len(certM
202     for i, asn1Data := range certMsg.certificate
203         if certs[i], err = x509.ParseCertifi
204             c.sendAlert(alertBadCertific
205             return errors.New("tls: fail
206         }
207     }
208
209     if c.config.ClientAuth >= VerifyClientCertIf
210         opts := x509.VerifyOptions{
211             Roots:          c.config.Clie
212             CurrentTime:    c.config.time
213             Intermediates: x509.NewCertP
214         }
215
216     for i, cert := range certs {
217         if i == 0 {
218             continue
219         }
220         opts.Intermediates.AddCert(c
221     }
222
223     chains, err := certs[0].Verify(opts)
224     if err != nil {
225         c.sendAlert(alertBadCertific
226         return errors.New("tls: fail
227     }
228
229     ok := false
230     for _, ku := range certs[0].ExtKeyUs
231         if ku == x509.ExtKeyUsageCli
232             ok = true
233             break
234         }
235     }
236     if !ok {
237         c.sendAlert(alertHandshakeFa
238         return errors.New("tls: clie
239     }

```

```

240
241         c.verifiedChains = chains
242     }
243
244     if len(certs) > 0 {
245         if pub, ok = certs[0].PublicKey.(*rsa
246             return c.sendAlert(alertUnsu
247         }
248         c.peerCertificates = certs
249     }
250
251     msg, err = c.readHandshake()
252     if err != nil {
253         return err
254     }
255 }
256
257 // Get client key exchange
258 ckx, ok := msg.(*clientKeyExchangeMsg)
259 if !ok {
260     return c.sendAlert(alertUnexpectedMessage)
261 }
262 finishedHash.Write(ckx.marshal())
263
264 // If we received a client cert in response to our c
265 // the client will send us a certificateVerifyMsg in
266 // clientKeyExchangeMsg. This message is a MD5SHA1
267 // handshake-layer messages that is signed using the
268 // to the client's certificate. This allows us to ve
269 // possession of the private key of the certificate.
270 if len(c.peerCertificates) > 0 {
271     msg, err = c.readHandshake()
272     if err != nil {
273         return err
274     }
275     certVerify, ok := msg.(*certificateVerifyMsg)
276     if !ok {
277         return c.sendAlert(alertUnexpectedMe
278     }
279
280     digest := make([]byte, 0, 36)
281     digest = finishedHash.serverMD5.Sum(digest)
282     digest = finishedHash.serverSHA1.Sum(digest)
283     err = rsa.VerifyPKCS1v15(pub, crypto.MD5SHA1
284     if err != nil {
285         c.sendAlert(alertBadCertificate)
286         return errors.New("could not validat
287     }
288

```

```

289         finishedHash.Write(certVerify.marshal())
290     }
291
292     preMasterSecret, err := keyAgreement.processClientKe
293     if err != nil {
294         c.sendAlert(alertHandshakeFailure)
295         return err
296     }
297
298     masterSecret, clientMAC, serverMAC, clientKey, serve
299         keysFromPreMasterSecret(c.vers, preMasterSec
300
301     clientCipher := suite.cipher(clientKey, clientIV, tr
302     clientHash := suite.mac(c.vers, clientMAC)
303     c.in.prepareCipherSpec(c.vers, clientCipher, clientH
304     c.readRecord(recordTypeChangeCipherSpec)
305     if err := c.error(); err != nil {
306         return err
307     }
308
309     if hello.nextProtoNeg {
310         msg, err = c.readHandshake()
311         if err != nil {
312             return err
313         }
314         nextProto, ok := msg.(*nextProtoMsg)
315         if !ok {
316             return c.sendAlert(alertUnexpectedMe
317         }
318         finishedHash.Write(nextProto.marshal())
319         c.clientProtocol = nextProto.proto
320     }
321
322     msg, err = c.readHandshake()
323     if err != nil {
324         return err
325     }
326     clientFinished, ok := msg.(*finishedMsg)
327     if !ok {
328         return c.sendAlert(alertUnexpectedMessage)
329     }
330
331     verify := finishedHash.clientSum(masterSecret)
332     if len(verify) != len(clientFinished.verifyData) ||
333         subtle.ConstantTimeCompare(verify, clientFin
334         return c.sendAlert(alertHandshakeFailure)
335     }
336
337     finishedHash.Write(clientFinished.marshal())

```

```
338
339     serverCipher := suite.cipher(serverKey, serverIV, fa
340     serverHash := suite.mac(c.vers, serverMAC)
341     c.out.prepareCipherSpec(c.vers, serverCipher, server
342     c.writeRecord(recordTypeChangeCipherSpec, []byte{1})
343
344     finished := new(finishedMsg)
345     finished.verifyData = finishedHash.serverSum(masterS
346     c.writeRecord(recordTypeHandshake, finished.marshal(
347
348     c.handshakeComplete = true
349     c.cipherSuite = suite.id
350
351     return nil
352 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/tls/key_agreement.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tls
6
7 import (
8     "crypto"
9     "crypto/elliptic"
10    "crypto/md5"
11    "crypto/rsa"
12    "crypto/sha1"
13    "crypto/x509"
14    "errors"
15    "io"
16    "math/big"
17 )
18
19 // rsaKeyAgreement implements the standard TLS key agreement
20 // encrypts the pre-master secret to the server's public key
21 type rsaKeyAgreement struct{}
22
23 func (ka rsaKeyAgreement) generateServerKeyExchange(config *
24     return nil, nil
25 }
26
27 func (ka rsaKeyAgreement) processClientKeyExchange(config *C
28     preMasterSecret := make([]byte, 48)
29     _, err := io.ReadFull(config.rand(), preMasterSecret
30     if err != nil {
31         return nil, err
32     }
33
34     if len(ckx.ciphertext) < 2 {
35         return nil, errors.New("bad ClientKeyExchang
36     }
37
38     ciphertext := ckx.ciphertext
39     if version != versionSSL30 {
40         ciphertextLen := int(ckx.ciphertext[0])<<8 |
41         if ciphertextLen != len(ckx.ciphertext)-2 {
```

```

42             return nil, errors.New("bad ClientKe
43         }
44         ciphertext = ckx.ciphertext[2:]
45     }
46
47     err = rsa.DecryptPKCS1v15SessionKey(config.rand(), c
48     if err != nil {
49         return nil, err
50     }
51     // We don't check the version number in the premaste
52     // by checking it, we would leak information about t
53     // encrypted pre-master secret. Secondly, it provide
54     // benefit against a downgrade attack and some imple
55     // wrong version anyway. See the discussion at the e
56     // 7.4.7.1 of RFC 4346.
57     return preMasterSecret, nil
58 }
59
60 func (ka rsaKeyAgreement) processServerKeyExchange(config *C
61     return errors.New("unexpected ServerKeyExchange")
62 }
63
64 func (ka rsaKeyAgreement) generateClientKeyExchange(config *
65     preMasterSecret := make([]byte, 48)
66     preMasterSecret[0] = byte(clientHello.vers >> 8)
67     preMasterSecret[1] = byte(clientHello.vers)
68     _, err := io.ReadFull(config.rand(), preMasterSecret
69     if err != nil {
70         return nil, nil, err
71     }
72
73     encrypted, err := rsa.EncryptPKCS1v15(config.rand(),
74     if err != nil {
75         return nil, nil, err
76     }
77     ckx := new(clientKeyExchangeMsg)
78     ckx.ciphertext = make([]byte, len(encrypted)+2)
79     ckx.ciphertext[0] = byte(len(encrypted) >> 8)
80     ckx.ciphertext[1] = byte(len(encrypted))
81     copy(ckx.ciphertext[2:], encrypted)
82     return preMasterSecret, ckx, nil
83 }
84
85 // md5SHA1Hash implements TLS 1.0's hybrid hash function whi
86 // concatenation of an MD5 and SHA1 hash.
87 func md5SHA1Hash(slices ...[]byte) []byte {
88     md5sha1 := make([]byte, md5.Size+sha1.Size)
89     hmd5 := md5.New()
90     for _, slice := range slices {
91         hmd5.Write(slice)

```

```

92     }
93     copy(md5sha1, hmd5.Sum(nil))
94
95     hsha1 := sha1.New()
96     for _, slice := range slices {
97         hsha1.Write(slice)
98     }
99     copy(md5sha1[md5.Size:], hsha1.Sum(nil))
100    return md5sha1
101 }
102
103 // ecdheRSAKeyAgreement implements a TLS key agreement where
104 // generates a ephemeral EC public/private key pair and sign
105 // pre-master secret is then calculated using ECDH.
106 type ecdheRSAKeyAgreement struct {
107     privateKey []byte
108     curve      elliptic.Curve
109     x, y       *big.Int
110 }
111
112 func (ka *ecdheRSAKeyAgreement) generateServerKeyExchange(co
113     var curveid uint16
114
115     Curve:
116     for _, c := range clientHello.supportedCurves {
117         switch c {
118             case curveP256:
119                 ka.curve = elliptic.P256()
120                 curveid = c
121                 break Curve
122             case curveP384:
123                 ka.curve = elliptic.P384()
124                 curveid = c
125                 break Curve
126             case curveP521:
127                 ka.curve = elliptic.P521()
128                 curveid = c
129                 break Curve
130         }
131     }
132
133     if curveid == 0 {
134         return nil, errors.New("tls: no supported el
135     }
136
137     var x, y *big.Int
138     var err error
139     ka.privateKey, x, y, err = elliptic.GenerateKey(ka.c
140     if err != nil {

```

```

141         return nil, err
142     }
143     ecdhePublic := elliptic.Marshal(ka.curve, x, y)
144
145     // http://tools.ietf.org/html/rfc4492#section-5.4
146     serverECDHParams := make([]byte, 1+2+1+len(ecdhePubl
147     serverECDHParams[0] = 3 // named curve
148     serverECDHParams[1] = byte(curveid >> 8)
149     serverECDHParams[2] = byte(curveid)
150     serverECDHParams[3] = byte(len(ecdhePublic))
151     copy(serverECDHParams[4:], ecdhePublic)
152
153     md5sha1 := md5SHA1Hash(clientHello.random, hello.ran
154     sig, err := rsa.SignPKCS1v15(config.rand(), cert.Pri
155     if err != nil {
156         return nil, errors.New("failed to sign ECDHE
157     }
158
159     skx := new(serverKeyExchangeMsg)
160     skx.key = make([]byte, len(serverECDHParams)+2+len(s
161     copy(skx.key, serverECDHParams)
162     k := skx.key[len(serverECDHParams):]
163     k[0] = byte(len(sig) >> 8)
164     k[1] = byte(len(sig))
165     copy(k[2:], sig)
166
167     return skx, nil
168 }
169
170 func (ka *ecdheRSAKeyAgreement) processClientKeyExchange(con
171     if len(ckx.ciphertext) == 0 || int(ckx.ciphertext[0]
172         return nil, errors.New("bad ClientKeyExchang
173     }
174     x, y := elliptic.Unmarshal(ka.curve, ckx.ciphertext[
175     if x == nil {
176         return nil, errors.New("bad ClientKeyExchang
177     }
178     x, _ = ka.curve.ScalarMult(x, y, ka.privateKey)
179     preMasterSecret := make([]byte, (ka.curve.Params()).B
180     xBytes := x.Bytes()
181     copy(preMasterSecret[len(preMasterSecret)-len(xBytes
182
183     return preMasterSecret, nil
184 }
185
186 var errServerKeyExchange = errors.New("invalid ServerKeyExch
187
188 func (ka *ecdheRSAKeyAgreement) processServerKeyExchange(con
189     if len(skx.key) < 4 {

```

```

190         return errServerKeyExchange
191     }
192     if skx.key[0] != 3 { // named curve
193         return errors.New("server selected unsupported
194     }
195     curveid := uint16(skx.key[1])<<8 | uint16(skx.key[2]
196
197     switch curveid {
198     case curveP256:
199         ka.curve = elliptic.P256()
200     case curveP384:
201         ka.curve = elliptic.P384()
202     case curveP521:
203         ka.curve = elliptic.P521()
204     default:
205         return errors.New("server selected unsupported
206     }
207
208     publicLen := int(skx.key[3])
209     if publicLen+4 > len(skx.key) {
210         return errServerKeyExchange
211     }
212     ka.x, ka.y = elliptic.Unmarshal(ka.curve, skx.key[4:
213     if ka.x == nil {
214         return errServerKeyExchange
215     }
216     serverECDHParams := skx.key[:4+publicLen]
217
218     sig := skx.key[4+publicLen:]
219     if len(sig) < 2 {
220         return errServerKeyExchange
221     }
222     sigLen := int(sig[0])<<8 | int(sig[1])
223     if sigLen+2 != len(sig) {
224         return errServerKeyExchange
225     }
226     sig = sig[2:]
227
228     md5sha1 := md5SHA1Hash(clientHello.random, serverHel
229     return rsa.VerifyPKCS1v15(cert.PublicKey.(*rsa.Publi
230 }
231
232 func (ka *ecdheRSAKeyAgreement) generateClientKeyExchange(co
233     if ka.curve == nil {
234         return nil, nil, errors.New("missing ServerK
235     }
236     priv, mx, my, err := elliptic.GenerateKey(ka.curve,
237     if err != nil {
238         return nil, nil, err
239     }

```

```
240     x, _ := ka.curve.ScalarMult(ka.x, ka.y, priv)
241     preMasterSecret := make([]byte, (ka.curve.Params().B
242     xBytes := x.Bytes()
243     copy(preMasterSecret[len(preMasterSecret)-len(xBytes)
244
245     serialized := elliptic.Marshal(ka.curve, mx, my)
246
247     ckx := new(clientKeyExchangeMsg)
248     ckx.ciphertext = make([]byte, 1+len(serialized))
249     ckx.ciphertext[0] = byte(len(serialized))
250     copy(ckx.ciphertext[1:], serialized)
251
252     return preMasterSecret, ckx, nil
253 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/crypto/tls/prf.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package tls
6
7 import (
8     "crypto/hmac"
9     "crypto/md5"
10    "crypto/sha1"
11    "hash"
12 )
13
14 // Split a premaster secret in two as specified in RFC 4346,
15 func splitPreMasterSecret(secret []byte) (s1, s2 []byte) {
16     s1 = secret[0 : (len(secret)+1)/2]
17     s2 = secret[len(secret)/2:]
18     return
19 }
20
21 // pHash implements the P_hash function, as defined in RFC 4
22 func pHash(result, secret, seed []byte, hash func() hash.Hash) {
23     h := hmac.New(hash, secret)
24     h.Write(seed)
25     a := h.Sum(nil)
26
27     j := 0
28     for j < len(result) {
29         h.Reset()
30         h.Write(a)
31         h.Write(seed)
32         b := h.Sum(nil)
33         todo := len(b)
34         if j+todo > len(result) {
35             todo = len(result) - j
36         }
37         copy(result[j:j+todo], b)
38         j += todo
39
40         h.Reset()
41         h.Write(a)
42         a = h.Sum(nil)
43     }
44 }
```

```

45
46 // pRF10 implements the TLS 1.0 pseudo-random function, as d
47 func pRF10(result, secret, label, seed []byte) {
48     hashSHA1 := sha1.New
49     hashMD5 := md5.New
50
51     labelAndSeed := make([]byte, len(label)+len(seed))
52     copy(labelAndSeed, label)
53     copy(labelAndSeed[len(label):], seed)
54
55     s1, s2 := splitPreMasterSecret(secret)
56     pHash(result, s1, labelAndSeed, hashMD5)
57     result2 := make([]byte, len(result))
58     pHash(result2, s2, labelAndSeed, hashSHA1)
59
60     for i, b := range result2 {
61         result[i] ^= b
62     }
63 }
64
65 // pRF30 implements the SSL 3.0 pseudo-random function, as d
66 // www.mozilla.org/projects/security/pki/nss/ssl/draft302.tx
67 func pRF30(result, secret, label, seed []byte) {
68     hashSHA1 := sha1.New()
69     hashMD5 := md5.New()
70
71     done := 0
72     i := 0
73     // RFC5246 section 6.3 says that the largest PRF out
74     // bytes. Since no more ciphersuites will be added t
75     // remain true. Each iteration gives us 16 bytes so
76     // be sufficient.
77     var b [16]byte
78     for done < len(result) {
79         for j := 0; j <= 15; j++ {
80             b[j] = 'A' + byte(i)
81         }
82
83         hashSHA1.Reset()
84         hashSHA1.Write(b[:16])
85         hashSHA1.Write(secret)
86         hashSHA1.Write(seed)
87         digest := hashSHA1.Sum(nil)
88
89         hashMD5.Reset()
90         hashMD5.Write(secret)
91         hashMD5.Write(digest)
92
93         done += copy(result[done:], hashMD5.Sum(nil))
94         i++

```

```

95     }
96 }
97
98 const (
99     tlsRandomLength      = 32 // Length of a random nonc
100     masterSecretLength  = 48 // Length of a master secr
101     finishedVerifyLength = 12 // Length of verify_data i
102 )
103
104 var masterSecretLabel = []byte("master secret")
105 var keyExpansionLabel = []byte("key expansion")
106 var clientFinishedLabel = []byte("client finished")
107 var serverFinishedLabel = []byte("server finished")
108
109 // keysFromPreMasterSecret generates the connection keys from
110 // secret, given the lengths of the MAC key, cipher key and
111 // RFC 2246, section 6.3.
112 func keysFromPreMasterSecret(version uint16, preMasterSecret
113     prf := pRF10
114     if version == versionSSL30 {
115         prf = pRF30
116     }
117
118     var seed [tlsRandomLength * 2]byte
119     copy(seed[0:len(clientRandom)], clientRandom)
120     copy(seed[len(clientRandom):], serverRandom)
121     masterSecret = make([]byte, masterSecretLength)
122     prf(masterSecret, preMasterSecret, masterSecretLabel)
123
124     copy(seed[0:len(clientRandom)], serverRandom)
125     copy(seed[len(serverRandom):], clientRandom)
126
127     n := 2*macLen + 2*keyLen + 2*ivLen
128     keyMaterial := make([]byte, n)
129     prf(keyMaterial, masterSecret, keyExpansionLabel, se
130     clientMAC = keyMaterial[:macLen]
131     keyMaterial = keyMaterial[macLen:]
132     serverMAC = keyMaterial[:macLen]
133     keyMaterial = keyMaterial[macLen:]
134     clientKey = keyMaterial[:keyLen]
135     keyMaterial = keyMaterial[keyLen:]
136     serverKey = keyMaterial[:keyLen]
137     keyMaterial = keyMaterial[keyLen:]
138     clientIV = keyMaterial[:ivLen]
139     keyMaterial = keyMaterial[ivLen:]
140     serverIV = keyMaterial[:ivLen]
141     return
142 }
143

```

```

144 func newFinishedHash(version uint16) finishedHash {
145     return finishedHash{md5.New(), sha1.New(), md5.New()}
146 }
147
148 // A finishedHash calculates the hash of a set of handshake
149 // for including in a Finished message.
150 type finishedHash struct {
151     clientMD5 hash.Hash
152     clientSHA1 hash.Hash
153     serverMD5 hash.Hash
154     serverSHA1 hash.Hash
155     version    uint16
156 }
157
158 func (h finishedHash) Write(msg []byte) (n int, err error) {
159     h.clientMD5.Write(msg)
160     h.clientSHA1.Write(msg)
161     h.serverMD5.Write(msg)
162     h.serverSHA1.Write(msg)
163     return len(msg), nil
164 }
165
166 // finishedSum10 calculates the contents of the verify_data
167 // Finished message given the MD5 and SHA1 hashes of a set o
168 // messages.
169 func finishedSum10(md5, sha1, label, masterSecret []byte) []
170     seed := make([]byte, len(md5)+len(sha1))
171     copy(seed, md5)
172     copy(seed[len(md5):], sha1)
173     out := make([]byte, finishedVerifyLength)
174     prf10(out, masterSecret, label, seed)
175     return out
176 }
177
178 // finishedSum30 calculates the contents of the verify_data
179 // Finished message given the MD5 and SHA1 hashes of a set o
180 // messages.
181 func finishedSum30(md5, sha1 hash.Hash, masterSecret []byte,
182     md5.Write(magic[:])
183     md5.Write(masterSecret)
184     md5.Write(ssl30Pad1[:])
185     md5Digest := md5.Sum(nil)
186
187     md5.Reset()
188     md5.Write(masterSecret)
189     md5.Write(ssl30Pad2[:])
190     md5.Write(md5Digest)
191     md5Digest = md5.Sum(nil)
192

```

```

193     sha1.Write(magic[:])
194     sha1.Write(masterSecret)
195     sha1.Write(ssl30Pad1[:40])
196     sha1Digest := sha1.Sum(nil)
197
198     sha1.Reset()
199     sha1.Write(masterSecret)
200     sha1.Write(ssl30Pad2[:40])
201     sha1.Write(sha1Digest)
202     sha1Digest = sha1.Sum(nil)
203
204     ret := make([]byte, len(md5Digest)+len(sha1Digest))
205     copy(ret, md5Digest)
206     copy(ret[len(md5Digest):], sha1Digest)
207     return ret
208 }
209
210 var ssl3ClientFinishedMagic = [4]byte{0x43, 0x4c, 0x4e, 0x54}
211 var ssl3ServerFinishedMagic = [4]byte{0x53, 0x52, 0x56, 0x52}
212
213 // clientSum returns the contents of the verify_data member
214 // Finished message.
215 func (h finishedHash) clientSum(masterSecret []byte) []byte
216     if h.version == versionSSL30 {
217         return finishedSum30(h.clientMD5, h.clientSH
218     }
219
220     md5 := h.clientMD5.Sum(nil)
221     sha1 := h.clientSHA1.Sum(nil)
222     return finishedSum10(md5, sha1, clientFinishedLabel,
223 }
224
225 // serverSum returns the contents of the verify_data member
226 // Finished message.
227 func (h finishedHash) serverSum(masterSecret []byte) []byte
228     if h.version == versionSSL30 {
229         return finishedSum30(h.serverMD5, h.serverSH
230     }
231
232     md5 := h.serverMD5.Sum(nil)
233     sha1 := h.serverSHA1.Sum(nil)
234     return finishedSum10(md5, sha1, serverFinishedLabel,
235 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/crypto/tls/tls.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package tls partially implements TLS 1.0, as specified in
6 package tls
7
8 import (
9     "crypto/rsa"
10    "crypto/x509"
11    "encoding/pem"
12    "errors"
13    "io/ioutil"
14    "net"
15    "strings"
16 )
17
18 // Server returns a new TLS server side connection
19 // using conn as the underlying transport.
20 // The configuration config must be non-nil and must have
21 // at least one certificate.
22 func Server(conn net.Conn, config *Config) *Conn {
23     return &Conn{conn: conn, config: config}
24 }
25
26 // Client returns a new TLS client side connection
27 // using conn as the underlying transport.
28 // Client interprets a nil configuration as equivalent to
29 // the zero configuration; see the documentation of Config
30 // for the defaults.
31 func Client(conn net.Conn, config *Config) *Conn {
32     return &Conn{conn: conn, config: config, isClient: true}
33 }
34
35 // A listener implements a network listener (net.Listener) for
36 type listener struct {
37     net.Listener
38     config *Config
39 }
40
41 // Accept waits for and returns the next incoming TLS connection.
42 // The returned connection c is a *tls.Conn.
43 func (l *listener) Accept() (c net.Conn, err error) {
44     c, err = l.Listener.Accept()
```

```

45         if err != nil {
46             return
47         }
48         c = Server(c, l.config)
49         return
50     }
51
52     // NewListener creates a Listener which accepts connections
53     // Listener and wraps each connection with Server.
54     // The configuration config must be non-nil and must have
55     // at least one certificate.
56     func NewListener(inner net.Listener, config *Config) net.Lis
57         l := new(listener)
58         l.Listener = inner
59         l.config = config
60         return l
61     }
62
63     // Listen creates a TLS listener accepting connections on th
64     // given network address using net.Listen.
65     // The configuration config must be non-nil and must have
66     // at least one certificate.
67     func Listen(network, laddr string, config *Config) (net.List
68         if config == nil || len(config.Certificates) == 0 {
69             return nil, errors.New("tls.Listen: no certi
70         }
71         l, err := net.Listen(network, laddr)
72         if err != nil {
73             return nil, err
74         }
75         return NewListener(l, config), nil
76     }
77
78     // Dial connects to the given network address using net.Dial
79     // and then initiates a TLS handshake, returning the resulti
80     // TLS connection.
81     // Dial interprets a nil configuration as equivalent to
82     // the zero configuration; see the documentation of Config
83     // for the defaults.
84     func Dial(network, addr string, config *Config) (*Conn, erro
85         raddr := addr
86         c, err := net.Dial(network, raddr)
87         if err != nil {
88             return nil, err
89         }
90
91         colonPos := strings.LastIndex(raddr, ":")
92         if colonPos == -1 {
93             colonPos = len(raddr)
94         }

```

```

95     hostname := raddr[:colonPos]
96
97     if config == nil {
98         config = defaultConfig()
99     }
100    // If no ServerName is set, infer the ServerName
101    // from the hostname we're connecting to.
102    if config.ServerName == "" {
103        // Make a copy to avoid polluting argument c
104        c := *config
105        c.ServerName = hostname
106        config = &c
107    }
108    conn := Client(c, config)
109    if err = conn.Handshake(); err != nil {
110        c.Close()
111        return nil, err
112    }
113    return conn, nil
114 }
115
116 // LoadX509KeyPair reads and parses a public/private key pair
117 // files. The files must contain PEM encoded data.
118 func LoadX509KeyPair(certFile, keyFile string) (cert Certifi
119     certPEMBlock, err := ioutil.ReadFile(certFile)
120     if err != nil {
121         return
122     }
123     keyPEMBlock, err := ioutil.ReadFile(keyFile)
124     if err != nil {
125         return
126     }
127     return X509KeyPair(certPEMBlock, keyPEMBlock)
128 }
129
130 // X509KeyPair parses a public/private key pair from a pair
131 // PEM encoded data.
132 func X509KeyPair(certPEMBlock, keyPEMBlock []byte) (cert Cer
133     var certDERBlock *pem.Block
134     for {
135         certDERBlock, certPEMBlock = pem.Decode(cert
136         if certDERBlock == nil {
137             break
138         }
139         if certDERBlock.Type == "CERTIFICATE" {
140             cert.Certificate = append(cert.Certi
141         }
142     }
143

```

```

144     if len(cert.Certificate) == 0 {
145         err = errors.New("crypto/tls: failed to pars
146         return
147     }
148
149     keyDERBlock, _ := pem.Decode(keyPEMBlock)
150     if keyDERBlock == nil {
151         err = errors.New("crypto/tls: failed to pars
152         return
153     }
154
155     // OpenSSL 0.9.8 generates PKCS#1 private keys by de
156     // OpenSSL 1.0.0 generates PKCS#8 keys. We try both.
157     var key *rsa.PrivateKey
158     if key, err = x509.ParsePKCS1PrivateKey(keyDERBlock.
159         var privKey interface{}
160         if privKey, err = x509.ParsePKCS8PrivateKey(
161             err = errors.New("crypto/tls: failed
162             return
163         }
164
165         var ok bool
166         if key, ok = privKey.(*rsa.PrivateKey); !ok
167             err = errors.New("crypto/tls: found
168             return
169         }
170     }
171
172     cert.PrivateKey = key
173
174     // We don't need to parse the public key for TLS, bu
175     // to check that it looks sane and matches the priva
176     x509Cert, err := x509.ParseCertificate(cert.Certific
177     if err != nil {
178         return
179     }
180
181     if x509Cert.PublicKeyAlgorithm != x509.RSA || x509Ce
182         err = errors.New("crypto/tls: private key do
183         return
184     }
185
186     return
187 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/x509/cert_pool.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package x509
6
7 import (
8     "encoding/pem"
9 )
10
11 // CertPool is a set of certificates.
12 type CertPool struct {
13     bySubjectKeyId map[string][]int
14     byName         map[string][]int
15     certs         []*Certificate
16 }
17
18 // NewCertPool returns a new, empty CertPool.
19 func NewCertPool() *CertPool {
20     return &CertPool{
21         make(map[string][]int),
22         make(map[string][]int),
23         nil,
24     }
25 }
26
27 // findVerifiedParents attempts to find certificates in s wh
28 // given certificate. If no such certificate can be found or
29 // doesn't match, it returns nil.
30 func (s *CertPool) findVerifiedParents(cert *Certificate) (p
31     if s == nil {
32         return
33     }
34     var candidates []int
35
36     if len(cert.AuthorityKeyId) > 0 {
37         candidates = s.bySubjectKeyId[string(cert.Au
38     }
39     if len(candidates) == 0 {
40         candidates = s.byName[string(cert.RawIssuer)
41     }
```

```

42
43     for _, c := range candidates {
44         if cert.CheckSignatureFrom(s.certs[c]) == nil {
45             parents = append(parents, c)
46         }
47     }
48
49     return
50 }
51
52 // AddCert adds a certificate to a pool.
53 func (s *CertPool) AddCert(cert *Certificate) {
54     if cert == nil {
55         panic("adding nil Certificate to CertPool")
56     }
57
58     // Check that the certificate isn't being added twice
59     for _, c := range s.certs {
60         if c.Equal(cert) {
61             return
62         }
63     }
64
65     n := len(s.certs)
66     s.certs = append(s.certs, cert)
67
68     if len(cert.SubjectKeyId) > 0 {
69         keyId := string(cert.SubjectKeyId)
70         s.bySubjectKeyId[keyId] = append(s.bySubject
71     }
72     name := string(cert.RawSubject)
73     s.byName[name] = append(s.byName[name], n)
74 }
75
76 // AppendCertsFromPEM attempts to parse a series of PEM encoded
77 // certificates found to s and returns true if
78 // were successfully parsed.
79 //
80 // On many Linux systems, /etc/ssl/cert.pem will contain the
81 // concatenation of root CAs in a format suitable for this function.
82 func (s *CertPool) AppendCertsFromPEM(pemCerts []byte) (ok bool) {
83     for len(pemCerts) > 0 {
84         var block *pem.Block
85         block, pemCerts = pem.Decode(pemCerts)
86         if block == nil {
87             break
88         }
89         if block.Type != "CERTIFICATE" || len(block.
90             continue
91     }

```

```
92
93         cert, err := ParseCertificate(block.Bytes)
94         if err != nil {
95             continue
96         }
97
98         s.AddCert(cert)
99         ok = true
100     }
101
102     return
103 }
104
105 // Subjects returns a list of the DER-encoded subjects of
106 // all of the certificates in the pool.
107 func (s *CertPool) Subjects() (res [][]byte) {
108     res = make([][]byte, len(s.certs))
109     for i, c := range s.certs {
110         res[i] = c.RawSubject
111     }
112     return
113 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/x509/pkcs1.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package x509
6
7 import (
8     "crypto/rsa"
9     "encoding/asn1"
10    "errors"
11    "math/big"
12 )
13
14 // pkcs1PrivateKey is a structure which mirrors the PKCS#1 A
15 type pkcs1PrivateKey struct {
16     Version int
17     N        *big.Int
18     E        int
19     D        *big.Int
20     P        *big.Int
21     Q        *big.Int
22     // We ignore these values, if present, because rsa w
23     Dp      *big.Int `asn1:"optional"`
24     Dq      *big.Int `asn1:"optional"`
25     Qinv    *big.Int `asn1:"optional"`
26
27     AdditionalPrimes []pkcs1AdditionalRSAPrime `asn1:"op
28 }
29
30 type pkcs1AdditionalRSAPrime struct {
31     Prime *big.Int
32
33     // We ignore these values because rsa will calculate
34     Exp *big.Int
35     Coeff *big.Int
36 }
37
38 // ParsePKCS1PrivateKey returns an RSA private key from its
39 func ParsePKCS1PrivateKey(der []byte) (key *rsa.PrivateKey,
40     var priv pkcs1PrivateKey
41     rest, err := asn1.Unmarshal(der, &priv)
```

```

42     if len(rest) > 0 {
43         err = asn1.SyntaxError{Msg: "trailing data"}
44         return
45     }
46     if err != nil {
47         return
48     }
49
50     if priv.Version > 1 {
51         return nil, errors.New("x509: unsupported pr
52     }
53
54     if priv.N.Sign() <= 0 || priv.D.Sign() <= 0 || priv.
55         return nil, errors.New("private key contains
56     }
57
58     key = new(rsa.PrivateKey)
59     key.PublicKey = rsa.PublicKey{
60         E: priv.E,
61         N: priv.N,
62     }
63
64     key.D = priv.D
65     key.Primes = make([]*big.Int, 2+len(priv.AdditionalP
66     key.Primes[0] = priv.P
67     key.Primes[1] = priv.Q
68     for i, a := range priv.AdditionalPrimes {
69         if a.Prime.Sign() <= 0 {
70             return nil, errors.New("private key
71         }
72         key.Primes[i+2] = a.Prime
73         // We ignore the other two values because rs
74         // them as needed.
75     }
76
77     err = key.Validate()
78     if err != nil {
79         return nil, err
80     }
81     key.Precompute()
82
83     return
84 }
85
86 // MarshalPKCS1PrivateKey converts a private key to ASN.1 DE
87 func MarshalPKCS1PrivateKey(key *rsa.PrivateKey) []byte {
88     key.Precompute()
89
90     version := 0
91     if len(key.Primes) > 2 {

```

```

92         version = 1
93     }
94
95     priv := pkcs1PrivateKey{
96         Version: version,
97         N:       key.N,
98         E:       key.PublicKey.E,
99         D:       key.D,
100        P:       key.Primes[0],
101        Q:       key.Primes[1],
102        Dp:      key.Precomputed.Dp,
103        Dq:      key.Precomputed.Dq,
104        Qinv:    key.Precomputed.Qinv,
105    }
106
107    priv.AdditionalPrimes = make([]pkcs1AdditionalRSAPri
108    for i, values := range key.Precomputed.CRTValues {
109        priv.AdditionalPrimes[i].Prime = key.Primes[
110        priv.AdditionalPrimes[i].Exp = values.Exp
111        priv.AdditionalPrimes[i].Coeff = values.Coeff
112    }
113
114    b, _ := asn1.Marshal(priv)
115    return b
116 }
117
118 // rsaPublicKey reflects the ASN.1 structure of a PKCS#1 pub
119 type rsaPublicKey struct {
120     N *big.Int
121     E int
122 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/x509/pkcs8.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package x509
6
7 import (
8     "crypto/x509/pkix"
9     "encoding/asn1"
10    "errors"
11    "fmt"
12 )
13
14 // pkcs8 reflects an ASN.1, PKCS#8 PrivateKey. See
15 // ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-8/pkcs-8v1_2.asn.
16 type pkcs8 struct {
17     Version      int
18     Algo         pkix.AlgorithmIdentifier
19     PrivateKey []byte
20     // optional attributes omitted.
21 }
22
23 // ParsePKCS8PrivateKey parses an unencrypted, PKCS#8 private
24 // key. See http://www.rsa.com/rsalabs/node.asp?id=2130
25 func ParsePKCS8PrivateKey(der []byte) (key interface{}, err error) {
26     var privKey pkcs8
27     if _, err := asn1.Unmarshal(der, &privKey); err != nil {
28         return nil, err
29     }
30     switch {
31     case privKey.Algo.Algorithm.Equal(oidRSA):
32         key, err = ParsePKCS1PrivateKey(privKey.PrivateKey)
33         if err != nil {
34             return nil, errors.New("crypto/x509:
35         }
36         return key, nil
37     default:
38         return nil, fmt.Errorf("crypto/x509: PKCS#8
39     }
40
41     panic("unreachable")

```

42 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/x509/root.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package x509
6
7 import "sync"
8
9 var (
10     once          sync.Once
11     systemRoots *CertPool
12 )
13
14 func systemRootsPool() *CertPool {
15     once.Do(initSystemRoots)
16     return systemRoots
17 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/x509/root_unix.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build freebsd linux openbsd netbsd
6
7 package x509
8
9 import "io/ioutil"
10
11 // Possible certificate files; stop after finding one.
12 var certFiles = []string{
13     "/etc/ssl/certs/ca-certificates.crt", // Linux e
14     "/etc/pki/tls/certs/ca-bundle.crt",   // Fedora/
15     "/etc/ssl/ca-bundle.pem",             // OpenSUS
16     "/etc/ssl/cert.pem",                  // OpenBSD
17     "/usr/local/share/certs/ca-root-nss.crt", // FreeBSD
18 }
19
20 func (c *Certificate) systemVerify(opts *VerifyOptions) (cha
21     return nil, nil
22 }
23
24 func initSystemRoots() {
25     roots := NewCertPool()
26     for _, file := range certFiles {
27         data, err := ioutil.ReadFile(file)
28         if err == nil {
29             roots.AppendCertsFromPEM(data)
30             break
31         }
32     }
33
34     systemRoots = roots
35 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/x509/verify.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package x509
6
7 import (
8     "runtime"
9     "strings"
10    "time"
11    "unicode/utf8"
12 )
13
14 type InvalidReason int
15
16 const (
17     // NotAuthorizedToSign results when a certificate is
18     // which isn't marked as a CA certificate.
19     NotAuthorizedToSign InvalidReason = iota
20     // Expired results when a certificate has expired, b
21     // given in the VerifyOptions.
22     Expired
23     // CANotAuthorizedForThisName results when an intern
24     // certificate has a name constraint which doesn't i
25     // being checked.
26     CANotAuthorizedForThisName
27     // TooManyIntermediates results when a path length c
28     // violated.
29     TooManyIntermediates
30 )
31
32 // CertificateInvalidError results when an odd error occurs.
33 // library probably want to handle all these errors uniforml
34 type CertificateInvalidError struct {
35     Cert    *Certificate
36     Reason InvalidReason
37 }
38
39 func (e CertificateInvalidError) Error() string {
40     switch e.Reason {
41     case NotAuthorizedToSign:
```

```

42         return "x509: certificate is not authorized
43     case Expired:
44         return "x509: certificate has expired or is
45     case CAnotAuthorizedForThisName:
46         return "x509: a root or intermediate certifi
47     case TooManyIntermediates:
48         return "x509: too many intermediates for pat
49     }
50     return "x509: unknown error"
51 }
52
53 // HostnameError results when the set of authorized names do
54 // requested name.
55 type HostnameError struct {
56     Certificate *Certificate
57     Host        string
58 }
59
60 func (h HostnameError) Error() string {
61     var valid string
62     c := h.Certificate
63     if len(c.DNSNames) > 0 {
64         valid = strings.Join(c.DNSNames, ", ")
65     } else {
66         valid = c.Subject.CommonName
67     }
68     return "certificate is valid for " + valid + ", not
69 }
70
71 // UnknownAuthorityError results when the certificate issuer
72 type UnknownAuthorityError struct {
73     cert *Certificate
74 }
75
76 func (e UnknownAuthorityError) Error() string {
77     return "x509: certificate signed by unknown authorit
78 }
79
80 // VerifyOptions contains parameters for Certificate.Verify.
81 // because other PKIX verification APIs have ended up needin
82 type VerifyOptions struct {
83     DNSName        string
84     Intermediates *CertPool
85     Roots          *CertPool // if nil, the system roots
86     CurrentTime    time.Time // if zero, the current time
87 }
88
89 const (
90     leafCertificate = iota
91     intermediateCertificate

```

```

92         rootCertificate
93     )
94
95     // isValid performs validity checks on the c.
96     func (c *Certificate) isValid(certType int, currentChain []*
97         now := opts.CurrentTime
98         if now.IsZero() {
99             now = time.Now()
100        }
101        if now.Before(c.NotBefore) || now.After(c.NotAfter)
102            return CertificateInvalidError{c, Expired}
103    }
104
105    if len(c.PermittedDNSDomains) > 0 {
106        for _, domain := range c.PermittedDNSDomains
107            if opts.DNSName == domain ||
108                (strings.HasSuffix(opts.DNSName, domain) &&
109                    len(opts.DNSName) >=
110                        len(domain) + 1) {
111                continue
112            }
113        }
114        return CertificateInvalidError{c, CA
115    }
116 }
117
118 // KeyUsage status flags are ignored. From Engineering
119 // Gutmann: A European government CA marked its sign
120 // being valid for encryption only, but no-one noticed
121 // European CA marked its signature keys as not being
122 // signatures. A different CA marked its own trusted
123 // as being invalid for certificate signing. Another
124 // distributed a certificate to be used to encrypt data
125 // country's tax authority that was marked as only being
126 // digital signatures but not for encryption. Yet another
127 // the order of the bit flags in the keyUsage due to
128 // encoding endianness, essentially setting a random
129 // certificates that it issued. Another CA created a
130 // certificate by adding a certificate policy statement
131 // that the certificate had to be used strictly as specified
132 // keyUsage, and a keyUsage containing a flag indicating
133 // encryption key could only be used for Diffie-Hellman
134
135 if certType == intermediateCertificate && (!c.BasicConstraints
136     return CertificateInvalidError{c, NotAuthorized}
137 }
138
139 if c.BasicConstraintsValid && c.MaxPathLen >= 0 {
140     numIntermediates := len(currentChain) - 1

```

```

141         if numIntermediates > c.MaxPathLen {
142             return CertificateInvalidError{c, To
143         }
144     }
145
146     return nil
147 }
148
149 // Verify attempts to verify c by building one or more chain
150 // certificate in opts.Roots, using certificates in opts.Int
151 // needed. If successful, it returns one or more chains wher
152 // element of the chain is c and the last element is from op
153 //
154 // WARNING: this doesn't do any revocation checking.
155 func (c *Certificate) Verify(opts VerifyOptions) (chains [][]
156     // Use Windows's own verification and chain building
157     if opts.Roots == nil && runtime.GOOS == "windows" {
158         return c.systemVerify(&opts)
159     }
160
161     if opts.Roots == nil {
162         opts.Roots = systemRootsPool()
163     }
164
165     err = c.isValid(leafCertificate, nil, &opts)
166     if err != nil {
167         return
168     }
169
170     if len(opts.DNSName) > 0 {
171         err = c.VerifyHostname(opts.DNSName)
172         if err != nil {
173             return
174         }
175     }
176
177     return c.buildChains(make(map[int][][]*Certificate),
178 }
179
180 func appendToFreshChain(chain []*Certificate, cert *Certific
181     n := make([]*Certificate, len(chain)+1)
182     copy(n, chain)
183     n[len(chain)] = cert
184     return n
185 }
186
187 func (c *Certificate) buildChains(cache map[int][][]*Certifi
188     for _, rootNum := range opts.Roots.findVerifiedParen
189         root := opts.Roots.certs[rootNum]

```

```

190         err = root.isValid(rootCertificate, currentC
191         if err != nil {
192             continue
193         }
194         chains = append(chains, appendToFreshChain(c
195     }
196
197     nextIntermediate:
198         for _, intermediateNum := range opts.Intermediates.f
199             intermediate := opts.Intermediates.certs[int
200             for _, cert := range currentChain {
201                 if cert == intermediate {
202                     continue nextIntermediate
203                 }
204             }
205             err = intermediate.isValid(intermediateCerti
206             if err != nil {
207                 continue
208             }
209             var childChains [][]*Certificate
210             childChains, ok := cache[intermediateNum]
211             if !ok {
212                 childChains, err = intermediate.buil
213                 cache[intermediateNum] = childChains
214             }
215             chains = append(chains, childChains...)
216         }
217
218         if len(chains) > 0 {
219             err = nil
220         }
221
222         if len(chains) == 0 && err == nil {
223             err = UnknownAuthorityError{c}
224         }
225
226         return
227     }
228
229     func matchHostnames(pattern, host string) bool {
230         if len(pattern) == 0 || len(host) == 0 {
231             return false
232         }
233
234         patternParts := strings.Split(pattern, ".")
235         hostParts := strings.Split(host, ".")
236
237         if len(patternParts) != len(hostParts) {
238             return false
239         }

```

```

240
241     for i, patternPart := range patternParts {
242         if patternPart == "*" {
243             continue
244         }
245         if patternPart != hostParts[i] {
246             return false
247         }
248     }
249
250     return true
251 }
252
253 // toLowerCaseASCII returns a lower-case version of in. See
254 // an explicitly ASCII function to avoid any sharp corners r
255 // performing Unicode operations on DNS labels.
256 func toLowerCaseASCII(in string) string {
257     // If the string is already lower-case then there's
258     isAlreadyLowerCase := true
259     for _, c := range in {
260         if c == utf8.RuneError {
261             // If we get a UTF-8 error then ther
262             // upper-case ASCII bytes in the inv
263             isAlreadyLowerCase = false
264             break
265         }
266         if 'A' <= c && c <= 'Z' {
267             isAlreadyLowerCase = false
268             break
269         }
270     }
271
272     if isAlreadyLowerCase {
273         return in
274     }
275
276     out := []byte(in)
277     for i, c := range out {
278         if 'A' <= c && c <= 'Z' {
279             out[i] += 'a' - 'A'
280         }
281     }
282     return string(out)
283 }
284
285 // VerifyHostname returns nil if c is a valid certificate fo
286 // Otherwise it returns an error describing the mismatch.
287 func (c *Certificate) VerifyHostname(h string) error {
288     lowered := toLowerCaseASCII(h)

```

```
289
290     if len(c.DNSNames) > 0 {
291         for _, match := range c.DNSNames {
292             if matchHostnames(toLowerCaseASCII(m
293                 return nil
294             }
295         }
296         // If Subject Alt Name is given, we ignore t
297     } else if matchHostnames(toLowerCaseASCII(c.Subject.
298         return nil
299     }
300
301     return HostnameError{c, h}
302 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/x509/x509.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package x509 parses X.509-encoded keys and certificates.
6 package x509
7
8 import (
9     "bytes"
10    "crypto"
11    "crypto/dsa"
12    "crypto/rsa"
13    "crypto/sha1"
14    "crypto/x509/pkix"
15    "encoding/asn1"
16    "encoding/pem"
17    "errors"
18    "io"
19    "math/big"
20    "time"
21 )
22
23 // pkixPublicKey reflects a PKIX public key structure. See S
24 // in RFC 3280.
25 type pkixPublicKey struct {
26     Algo      pkix.AlgorithmIdentifier
27     BitString asn1.BitString
28 }
29
30 // ParsePKIXPublicKey parses a DER encoded public key. These
31 // typically found in PEM blocks with "BEGIN PUBLIC KEY".
32 func ParsePKIXPublicKey(derBytes []byte) (pub interface{}, e
33     var pki publicKeyInfo
34     if _, err = asn1.Unmarshal(derBytes, &pki); err != n
35         return
36     }
37     algo := getPublicKeyAlgorithmFromOID(pki.Algorithm.A
38     if algo == UnknownPublicKeyAlgorithm {
39         return nil, errors.New("ParsePKIXPublicKey:
40     }
41     return parsePublicKey(algo, &pki)
```

```

42 }
43
44 // MarshalPKIXPublicKey serialises a public key to DER-encod
45 func MarshalPKIXPublicKey(pub interface{}) ([]byte, error) {
46     var pubBytes []byte
47
48     switch pub := pub.(type) {
49     case *rsa.PublicKey:
50         pubBytes, _ = asn1.Marshal(rsaPublicKey{
51             N: pub.N,
52             E: pub.E,
53         })
54     default:
55         return nil, errors.New("MarshalPKIXPublicKey
56     }
57
58     pkix := pkixPublicKey{
59         Algo: pkix.AlgorithmIdentifier{
60             Algorithm: []int{1, 2, 840, 113549,
61             // This is a NULL parameters value w
62             // superfluous, but most other code
63             // doing this, we match their public
64             Parameters: asn1.RawValue{
65                 Tag: 5,
66             },
67         },
68         BitString: asn1.BitString{
69             Bytes: pubBytes,
70             BitLength: 8 * len(pubBytes),
71         },
72     }
73
74     ret, _ := asn1.Marshal(pkix)
75     return ret, nil
76 }
77
78 // These structures reflect the ASN.1 structure of X.509 cer
79
80 type certificate struct {
81     Raw          asn1.RawContent
82     TBSCertificate tbsCertificate
83     SignatureAlgorithm pkix.AlgorithmIdentifier
84     SignatureValue  asn1.BitString
85 }
86
87 type tbsCertificate struct {
88     Raw          asn1.RawContent
89     Version      int `asn1:"optional,explicit,defa
90     SerialNumber *big.Int
91     SignatureAlgorithm pkix.AlgorithmIdentifier

```

```

92         Issuer          asn1.RawValue
93         Validity        validity
94         Subject         asn1.RawValue
95         PublicKey       publicKeyInfo
96         UniqueId        asn1.BitString `asn1:"optional,
97         SubjectUniqueId asn1.BitString `asn1:"optional,
98         Extensions     []pkix.Extension `asn1:"optional,
99     }
100
101     type dsaAlgorithmParameters struct {
102         P, Q, G *big.Int
103     }
104
105     type dsaSignature struct {
106         R, S *big.Int
107     }
108
109     type validity struct {
110         NotBefore, NotAfter time.Time
111     }
112
113     type publicKeyInfo struct {
114         Raw          asn1.RawContent
115         Algorithm    pkix.AlgorithmIdentifier
116         PublicKey    asn1.BitString
117     }
118
119     // RFC 5280, 4.2.1.1
120     type authKeyId struct {
121         Id []byte `asn1:"optional,tag:0"`
122     }
123
124     type SignatureAlgorithm int
125
126     const (
127         UnknownSignatureAlgorithm SignatureAlgorithm = iota
128         MD2WithRSA
129         MD5WithRSA
130         SHA1WithRSA
131         SHA256WithRSA
132         SHA384WithRSA
133         SHA512WithRSA
134         DSAWithSHA1
135         DSAWithSHA256
136     )
137
138     type PublicKeyAlgorithm int
139
140     const (

```

```

141         UnknownPublicKeyAlgorithm PublicKeyAlgorithm = iota
142         RSA
143         DSA
144     )
145
146     // OIDs for signature algorithms
147     //
148     // pkcs-1 OBJECT IDENTIFIER ::= {
149     //     iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) 1
150     //
151     //
152     // RFC 3279 2.2.1 RSA Signature Algorithms
153     //
154     // md2WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 2 }
155     //
156     // md5WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 4 }
157     //
158     // sha-1WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 5 }
159     //
160     // dsaWithSha1 OBJECT IDENTIFIER ::= {
161     //     iso(1) member-body(2) us(840) x9-57(10040) x9cm(4) 3 }
162     //
163     //
164     // RFC 4055 5 PKCS #1 Version 1.5
165     //
166     // sha256WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 11
167     //
168     // sha384WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 12
169     //
170     // sha512WithRSAEncryption OBJECT IDENTIFIER ::= { pkcs-1 13
171     //
172     //
173     // RFC 5758 3.1 DSA Signature Algorithms
174     //
175     // dsaWithSha256 OBJECT IDENTIFIER ::= {
176     //     joint-iso-ccitt(2) country(16) us(840) organization(1)
177     //     csor(3) algorithms(4) id-dsa-with-sha2(3) 2}
178     //
179     var (
180         oidSignatureMD2WithRSA      = asn1.ObjectIdentifier{1,
181         oidSignatureMD5WithRSA      = asn1.ObjectIdentifier{1,
182         oidSignatureSHA1WithRSA     = asn1.ObjectIdentifier{1,
183         oidSignatureSHA256WithRSA   = asn1.ObjectIdentifier{1,
184         oidSignatureSHA384WithRSA   = asn1.ObjectIdentifier{1,
185         oidSignatureSHA512WithRSA   = asn1.ObjectIdentifier{1,
186         oidSignatureDSAWithSHA1     = asn1.ObjectIdentifier{1,
187         oidSignatureDSAWithSHA256  = asn1.ObjectIdentifier{2,
188     )
189

```

```

190 func getSignatureAlgorithmFromOID(oid asn1.ObjectIdentifier)
191     switch {
192     case oid.Equal(oidSignatureMD2WithRSA):
193         return MD2WithRSA
194     case oid.Equal(oidSignatureMD5WithRSA):
195         return MD5WithRSA
196     case oid.Equal(oidSignatureSHA1WithRSA):
197         return SHA1WithRSA
198     case oid.Equal(oidSignatureSHA256WithRSA):
199         return SHA256WithRSA
200     case oid.Equal(oidSignatureSHA384WithRSA):
201         return SHA384WithRSA
202     case oid.Equal(oidSignatureSHA512WithRSA):
203         return SHA512WithRSA
204     case oid.Equal(oidSignatureDSAWithSHA1):
205         return DSAWithSHA1
206     case oid.Equal(oidSignatureDSAWithSHA256):
207         return DSAWithSHA256
208     }
209     return UnknownSignatureAlgorithm
210 }
211
212 // RFC 3279, 2.3 Public Key Algorithms
213 //
214 // pkcs-1 OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(
215 //     rsadsi(113549) pkcs(1) 1 }
216 //
217 // rsaEncryption OBJECT IDENTIFIER ::= { pkcs1-1 1 }
218 //
219 // id-dsa OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(
220 //     x9-57(10040) x9cm(4) 1 }
221 var (
222     oidPublicKeyRsa = asn1.ObjectIdentifier{1, 2, 840, 1
223     oidPublicKeyDsa = asn1.ObjectIdentifier{1, 2, 840, 1
224 )
225
226 func getPublicKeyAlgorithmFromOID(oid asn1.ObjectIdentifier)
227     switch {
228     case oid.Equal(oidPublicKeyRsa):
229         return RSA
230     case oid.Equal(oidPublicKeyDsa):
231         return DSA
232     }
233     return UnknownPublicKeyAlgorithm
234 }
235
236 // KeyUsage represents the set of actions that are valid for
237 // a bitmap of the KeyUsage* constants.
238 type KeyUsage int
239

```

```

240 const (
241     KeyUsageDigitalSignature KeyUsage = 1 << iota
242     KeyUsageContentCommitment
243     KeyUsageKeyEncipherment
244     KeyUsageDataEncipherment
245     KeyUsageKeyAgreement
246     KeyUsageCertSign
247     KeyUsageCRLSign
248     KeyUsageEncipherOnly
249     KeyUsageDecipherOnly
250 )
251
252 // RFC 5280, 4.2.1.12 Extended Key Usage
253 //
254 // anyExtendedKeyUsage OBJECT IDENTIFIER ::= { id-ce-extKeyU
255 //
256 // id-kp OBJECT IDENTIFIER ::= { id-pkix 3 }
257 //
258 // id-kp-serverAuth          OBJECT IDENTIFIER ::= { id-k
259 // id-kp-clientAuth          OBJECT IDENTIFIER ::= { id-k
260 // id-kp-codeSigning         OBJECT IDENTIFIER ::= { id-k
261 // id-kp-emailProtection     OBJECT IDENTIFIER ::= { id-k
262 // id-kp-timeStamping        OBJECT IDENTIFIER ::= { id-k
263 // id-kp-OCSPSigning         OBJECT IDENTIFIER ::= { id-k
264 var (
265     oidExtKeyUsageAny          = asn1.ObjectIdentifie
266     oidExtKeyUsageServerAuth   = asn1.ObjectIdentifie
267     oidExtKeyUsageClientAuth   = asn1.ObjectIdentifie
268     oidExtKeyUsageCodeSigning  = asn1.ObjectIdentifie
269     oidExtKeyUsageEmailProtection = asn1.ObjectIdentifie
270     oidExtKeyUsageTimeStamping = asn1.ObjectIdentifie
271     oidExtKeyUsageOCSPSigning  = asn1.ObjectIdentifie
272 )
273
274 // ExtKeyUsage represents an extended set of actions that ar
275 // Each of the ExtKeyUsage* constants define a unique action
276 type ExtKeyUsage int
277
278 const (
279     ExtKeyUsageAny ExtKeyUsage = iota
280     ExtKeyUsageServerAuth
281     ExtKeyUsageClientAuth
282     ExtKeyUsageCodeSigning
283     ExtKeyUsageEmailProtection
284     ExtKeyUsageTimeStamping
285     ExtKeyUsageOCSPSigning
286 )
287
288 // A Certificate represents an X.509 certificate.

```

```

289 type Certificate struct {
290     Raw []byte // Complete ASN.1 DER
291     RawTBSCertificate []byte // Certificate part 0
292     RawSubjectPublicKeyInfo []byte // DER encoded Subjec
293     RawSubject []byte // DER encoded Subjec
294     RawIssuer []byte // DER encoded Issuer
295
296     Signature []byte
297     SignatureAlgorithm SignatureAlgorithm
298
299     PublicKeyAlgorithm PublicKeyAlgorithm
300     PublicKey interface{}
301
302     Version int
303     SerialNumber *big.Int
304     Issuer pkix.Name
305     Subject pkix.Name
306     NotBefore, NotAfter time.Time // Validity bounds.
307     KeyUsage KeyUsage
308
309     ExtKeyUsage []ExtKeyUsage // Sequen
310     UnknownExtKeyUsage []asn1.ObjectIdentifier // Encoun
311
312     BasicConstraintsValid bool // if true then the next
313     IsCA bool
314     MaxPathLen int
315
316     SubjectKeyId []byte
317     AuthorityKeyId []byte
318
319     // Subject Alternate Name values
320     DNSNames []string
321     EmailAddresses []string
322
323     // Name constraints
324     PermittedDNSDomainsCritical bool // if true then the
325     PermittedDNSDomains []string
326
327     PolicyIdentifiers []asn1.ObjectIdentifier
328 }
329
330 // ErrUnsupportedAlgorithm results from attempting to perfor
331 // involves algorithms that are not currently implemented.
332 var ErrUnsupportedAlgorithm = errors.New("crypto/x509: canno
333
334 // ConstraintViolationError results when a requested usage i
335 // a certificate. For example: checking a signature when the
336 // certificate signing key.
337 type ConstraintViolationError struct{}

```

```

338
339 func (ConstraintViolationError) Error() string {
340     return "crypto/x509: invalid signature: parent certi
341 }
342
343 func (c *Certificate) Equal(other *Certificate) bool {
344     return bytes.Equal(c.Raw, other.Raw)
345 }
346
347 // CheckSignatureFrom verifies that the signature on c is a
348 // from parent.
349 func (c *Certificate) CheckSignatureFrom(parent *Certificate
350     // RFC 5280, 4.2.1.9:
351     // "If the basic constraints extension is not presen
352     // certificate, or the extension is present but the
353     // asserted, then the certified public key MUST NOT
354     // certificate signatures."
355     if parent.Version == 3 && !parent.BasicConstraintsVa
356         parent.BasicConstraintsValid && !parent.IsCA
357         return ConstraintViolationError{}
358     }
359
360     if parent.KeyUsage != 0 && parent.KeyUsage&KeyUsageC
361         return ConstraintViolationError{}
362     }
363
364     if parent.PublicKeyAlgorithm == UnknownPublicKeyAlgo
365         return ErrUnsupportedAlgorithm
366     }
367
368     // TODO(agl): don't ignore the path length constrain
369
370     return parent.CheckSignature(c.SignatureAlgorithm, c
371 }
372
373 // CheckSignature verifies that signature is a valid signatu
374 // c's public key.
375 func (c *Certificate) CheckSignature(algo SignatureAlgorithr
376     var hashType crypto.Hash
377
378     switch algo {
379     case SHA1WithRSA, DSAWithSHA1:
380         hashType = crypto.SHA1
381     case SHA256WithRSA, DSAWithSHA256:
382         hashType = crypto.SHA256
383     case SHA384WithRSA:
384         hashType = crypto.SHA384
385     case SHA512WithRSA:
386         hashType = crypto.SHA512
387     default:

```

```

388         return ErrUnsupportedAlgorithm
389     }
390
391     h := hashType.New()
392     if h == nil {
393         return ErrUnsupportedAlgorithm
394     }
395
396     h.Write(signed)
397     digest := h.Sum(nil)
398
399     switch pub := c.PublicKey.(type) {
400     case *rsa.PublicKey:
401         return rsa.VerifyPKCS1v15(pub, hashType, dig
402     case *dsa.PublicKey:
403         dsaSig := new(dsaSignature)
404         if _, err := asn1.Unmarshal(signature, dsaSi
405             return err
406         }
407         if dsaSig.R.Sign() <= 0 || dsaSig.S.Sign() <
408             return errors.New("DSA signature con
409         }
410         if !dsa.Verify(pub, digest, dsaSig.R, dsaSig
411             return errors.New("DSA verification
412         }
413         return
414     }
415     return ErrUnsupportedAlgorithm
416 }
417
418 // CheckCRLSignature checks that the signature in crl is fro
419 func (c *Certificate) CheckCRLSignature(crl *pkix.Certificat
420     algo := getSignatureAlgorithmFromOID(crl.SignatureAl
421     return c.CheckSignature(algo, crl.TBSCertList.Raw, c
422 }
423
424 type UnhandledCriticalExtension struct{}
425
426 func (h UnhandledCriticalExtension) Error() string {
427     return "unhandled critical extension"
428 }
429
430 type basicConstraints struct {
431     IsCA          bool `asn1:"optional"`
432     MaxPathLen    int  `asn1:"optional,default:-1"`
433 }
434
435 // RFC 5280 4.2.1.4
436 type policyInformation struct {

```

```

437         Policy asn1.ObjectIdentifier
438         // policyQualifiers omitted
439     }
440
441     // RFC 5280, 4.2.1.10
442     type nameConstraints struct {
443         Permitted []generalSubtree `asn1:"optional,tag:0"`
444         Excluded []generalSubtree `asn1:"optional,tag:1"`
445     }
446
447     type generalSubtree struct {
448         Name string `asn1:"tag:2,optional,ia5"`
449         Min int    `asn1:"optional,tag:0"`
450         Max int    `asn1:"optional,tag:1"`
451     }
452
453     func parsePublicKey(algo PublicKeyAlgorithm, keyData *public
454         asn1Data := keyData.PublicKey.RightAlign()
455         switch algo {
456         case RSA:
457             p := new(rsaPublicKey)
458             _, err := asn1.Unmarshal(asn1Data, p)
459             if err != nil {
460                 return nil, err
461             }
462
463             pub := &rsa.PublicKey{
464                 E: p.E,
465                 N: p.N,
466             }
467             return pub, nil
468         case DSA:
469             var p *big.Int
470             _, err := asn1.Unmarshal(asn1Data, &p)
471             if err != nil {
472                 return nil, err
473             }
474             paramsData := keyData.Algorithm.Parameters.F
475             params := new(dsaAlgorithmParameters)
476             _, err = asn1.Unmarshal(paramsData, params)
477             if err != nil {
478                 return nil, err
479             }
480             if p.Sign() <= 0 || params.P.Sign() <= 0 ||
481                 return nil, errors.New("zero or nega
482             }
483             pub := &dsa.PublicKey{
484                 Parameters: dsa.Parameters{
485                     P: params.P,

```

```

486             Q: params.Q,
487             G: params.G,
488             },
489             Y: p,
490         }
491         return pub, nil
492     default:
493         return nil, nil
494     }
495     panic("unreachable")
496 }
497
498 func parseCertificate(in *certificate) (*Certificate, error)
499     out := new(Certificate)
500     out.Raw = in.Raw
501     out.RawTBSCertificate = in.TBSCertificate.Raw
502     out.RawSubjectPublicKeyInfo = in.TBSCertificate.Publ
503     out.RawSubject = in.TBSCertificate.Subject.FullBytes
504     out.RawIssuer = in.TBSCertificate.Issuer.FullBytes
505
506     out.Signature = in.SignatureValue.RightAlign()
507     out.SignatureAlgorithm =
508         getSignatureAlgorithmFromOID(in.TBSCertifica
509
510     out.PublicKeyAlgorithm =
511         getPublicKeyAlgorithmFromOID(in.TBSCertifica
512     var err error
513     out.PublicKey, err = parsePublicKey(out.PublicKeyAlg
514     if err != nil {
515         return nil, err
516     }
517
518     if in.TBSCertificate.SerialNumber.Sign() < 0 {
519         return nil, errors.New("negative serial numb
520     }
521
522     out.Version = in.TBSCertificate.Version + 1
523     out.SerialNumber = in.TBSCertificate.SerialNumber
524
525     var issuer, subject pkix.RDNSequence
526     if _, err := asn1.Unmarshal(in.TBSCertificate.Subjec
527         return nil, err
528     }
529     if _, err := asn1.Unmarshal(in.TBSCertificate.Issuer
530         return nil, err
531     }
532
533     out.Issuer.FillFromRDNSequence(&issuer)
534     out.Subject.FillFromRDNSequence(&subject)
535

```

```

536 out.NotBefore = in.TBSCertificate.Validity.NotBefore
537 out.NotAfter = in.TBSCertificate.Validity.NotAfter
538
539 for _, e := range in.TBSCertificate.Extensions {
540     if len(e.Id) == 4 && e.Id[0] == 2 && e.Id[1]
541         switch e.Id[3] {
542             case 15:
543                 // RFC 5280, 4.2.1.3
544                 var usageBits asn1.BitString
545                 _, err := asn1.Unmarshal(e.V
546
547                 if err == nil {
548                     var usage int
549                     for i := 0; i < 9; i
550                         if usageBits
551                             usag
552                     }
553                 }
554                 out.KeyUsage = KeyUs
555                 continue
556             }
557             case 19:
558                 // RFC 5280, 4.2.1.9
559                 var constraints basicConstra
560                 _, err := asn1.Unmarshal(e.V
561
562                 if err == nil {
563                     out.BasicConstraints
564                     out.IsCA = constrain
565                     out.MaxPathLen = con
566                     continue
567                 }
568             case 17:
569                 // RFC 5280, 4.2.1.6
570
571                 // SubjectAltName ::= Genera
572                 //
573                 // GeneralNames ::= SEQUENCE
574                 //
575                 // GeneralName ::= CHOICE {
576                 //     otherName
577                 //     rfc822Name
578                 //     dNSName
579                 //     x400Address
580                 //     directoryName
581                 //     ediPartyName
582                 //     uniformResourceIdent
583                 //     iPAddress
584                 //     registeredID

```

```

585     var seq asn1.RawValue
586     _, err := asn1.Unmarshal(e.v
587     if err != nil {
588         return nil, err
589     }
590     if !seq.IsCompound || seq.Tag
591         return nil, asn1.Str
592     }
593
594     parsedName := false
595
596     rest := seq.Bytes
597     for len(rest) > 0 {
598         var v asn1.RawValue
599         rest, err = asn1.Unr
600         if err != nil {
601             return nil,
602         }
603         switch v.Tag {
604         case 1:
605             out.EmailAdd
606             parsedName =
607         case 2:
608             out.DNSNames
609             parsedName =
610         }
611     }
612
613     if parsedName {
614         continue
615     }
616     // If we didn't parse any of
617     // fall through to the criti
618
619     case 30:
620         // RFC 5280, 4.2.1.10
621
622         // NameConstraints ::= SEQUE
623         //     permittedSubtrees
624         //     excludedSubtrees
625         //
626         // GeneralSubtrees ::= SEQUE
627         //
628         // GeneralSubtree ::= SEQUEN
629         //     base
630         //     minimum           [0]
631         //     maximum           [1]
632         //
633         // BaseDistance ::= INTEGER

```

```

634
635         var constraints nameConstrai
636         _, err := asn1.Unmarshal(e.V
637         if err != nil {
638             return nil, err
639         }
640
641         if len(constraints.Excluded)
642             return out, Unhandle
643     }
644
645     for _, subtree := range cons
646         if subtree.Min > 0 |
647             if e.Critica
648                 retu
649             }
650             continue
651     }
652     out.PermittedDNSDoma
653 }
654 continue
655
656 case 35:
657     // RFC 5280, 4.2.1.1
658     var a authKeyId
659     _, err = asn1.Unmarshal(e.Va
660     if err != nil {
661         return nil, err
662     }
663     out.AuthorityKeyId = a.Id
664     continue
665
666 case 37:
667     // RFC 5280, 4.2.1.12. Exte
668
669     // id-ce-extKeyUsage OBJECT
670     //
671     // ExtKeyUsageSyntax ::= SEQ
672     //
673     // KeyPurposeId ::= OBJECT I
674
675     var keyUsage []asn1.ObjectId
676     _, err = asn1.Unmarshal(e.Va
677     if err != nil {
678         return nil, err
679     }
680
681     for _, u := range keyUsage {
682         switch {
683             case u.Equal(oidExtK

```

```

684         out.ExtKeyUs
685     case u.Equal(oidExtK
686         out.ExtKeyUs
687     case u.Equal(oidExtK
688         out.ExtKeyUs
689     case u.Equal(oidExtK
690         out.ExtKeyUs
691     case u.Equal(oidExtK
692         out.ExtKeyUs
693     case u.Equal(oidExtK
694         out.ExtKeyUs
695     case u.Equal(oidExtK
696         out.ExtKeyUs
697     default:
698         out.UnknownE
699     }
700 }
701
702     continue
703
704     case 14:
705         // RFC 5280, 4.2.1.2
706         var keyid []byte
707         _, err = asn1.Unmarshal(e.Va
708         if err != nil {
709             return nil, err
710         }
711         out.SubjectKeyId = keyid
712         continue
713
714     case 32:
715         // RFC 5280 4.2.1.4: Certifi
716         var policies []policyInforma
717         if _, err = asn1.Unmarshal(e
718             return nil, err
719         }
720         out.PolicyIdentifiers = make
721         for i, policy := range polic
722             out.PolicyIdentifier
723         }
724     }
725 }
726
727     if e.Critical {
728         return out, UnhandledCriticalExtensi
729     }
730 }
731
732     return out, nil

```

```

733 }
734
735 // ParseCertificate parses a single certificate from the giv
736 func ParseCertificate(asn1Data []byte) (*Certificate, error)
737     var cert certificate
738     rest, err := asn1.Unmarshal(asn1Data, &cert)
739     if err != nil {
740         return nil, err
741     }
742     if len(rest) > 0 {
743         return nil, asn1.SyntaxError{Msg: "trailing
744     }
745
746     return parseCertificate(&cert)
747 }
748
749 // ParseCertificates parses one or more certificates from th
750 // data. The certificates must be concatenated with no inter
751 func ParseCertificates(asn1Data []byte) ([]*Certificate, err
752     var v []*certificate
753
754     for len(asn1Data) > 0 {
755         cert := new(certificate)
756         var err error
757         asn1Data, err = asn1.Unmarshal(asn1Data, cer
758         if err != nil {
759             return nil, err
760         }
761         v = append(v, cert)
762     }
763
764     ret := make([]*Certificate, len(v))
765     for i, ci := range v {
766         cert, err := parseCertificate(ci)
767         if err != nil {
768             return nil, err
769         }
770         ret[i] = cert
771     }
772
773     return ret, nil
774 }
775
776 func reverseBitsInAByte(in byte) byte {
777     b1 := in>>4 | in<<4
778     b2 := b1>>2&0x33 | b1<<2&0xcc
779     b3 := b2>>1&0x55 | b2<<1&0xaa
780     return b3
781 }

```

```

782
783 var (
784     oidExtensionSubjectKeyId          = []int{2, 5, 29, 14}
785     oidExtensionKeyUsage              = []int{2, 5, 29, 15}
786     oidExtensionAuthorityKeyId        = []int{2, 5, 29, 35}
787     oidExtensionBasicConstraints     = []int{2, 5, 29, 19}
788     oidExtensionSubjectAltName       = []int{2, 5, 29, 17}
789     oidExtensionCertificatePolicies  = []int{2, 5, 29, 32}
790     oidExtensionNameConstraints      = []int{2, 5, 29, 30}
791 )
792
793 func buildExtensions(template *Certificate) (ret []pkix.Extensio
794     ret = make([]pkix.Extension, 7 /* maximum number of
795     n := 0
796
797     if template.KeyUsage != 0 {
798         ret[n].Id = oidExtensionKeyUsage
799         ret[n].Critical = true
800
801         var a [2]byte
802         a[0] = reverseBitsInAByte(byte(template.KeyU
803         a[1] = reverseBitsInAByte(byte(template.KeyU
804
805         l := 1
806         if a[1] != 0 {
807             l = 2
808         }
809
810         ret[n].Value, err = asn1.Marshal(asn1.BitStr
811         if err != nil {
812             return
813         }
814         n++
815     }
816
817     if template.BasicConstraintsValid {
818         ret[n].Id = oidExtensionBasicConstraints
819         ret[n].Value, err = asn1.Marshal(basicConstr
820         ret[n].Critical = true
821         if err != nil {
822             return
823         }
824         n++
825     }
826
827     if len(template.SubjectKeyId) > 0 {
828         ret[n].Id = oidExtensionSubjectKeyId
829         ret[n].Value, err = asn1.Marshal(template.Su
830         if err != nil {
831             return

```

```

832         }
833         n++
834     }
835
836     if len(template.AuthorityKeyId) > 0 {
837         ret[n].Id = oidExtensionAuthorityKeyId
838         ret[n].Value, err = asn1.Marshal(authKeyId{t
839         if err != nil {
840             return
841         }
842         n++
843     }
844
845     if len(template.DNSNames) > 0 {
846         ret[n].Id = oidExtensionSubjectAltName
847         rawValues := make([]asn1.RawValue, len(templ
848         for i, name := range template.DNSNames {
849             rawValues[i] = asn1.RawValue{Tag: 2,
850         }
851         ret[n].Value, err = asn1.Marshal(rawValues)
852         if err != nil {
853             return
854         }
855         n++
856     }
857
858     if len(template.PolicyIdentifiers) > 0 {
859         ret[n].Id = oidExtensionCertificatePolicies
860         policies := make([]policyInformation, len(te
861         for i, policy := range template.PolicyIdenti
862             policies[i].Policy = policy
863         }
864         ret[n].Value, err = asn1.Marshal(policies)
865         if err != nil {
866             return
867         }
868         n++
869     }
870
871     if len(template.PermittedDNSDomains) > 0 {
872         ret[n].Id = oidExtensionNameConstraints
873         ret[n].Critical = template.PermittedDNSDomai
874
875         var out nameConstraints
876         out.Permitted = make([]generalSubtree, len(t
877         for i, permitted := range template.Permitted
878             out.Permitted[i] = generalSubtree{Na
879         }
880         ret[n].Value, err = asn1.Marshal(out)

```

```

881         if err != nil {
882             return
883         }
884         n++
885     }
886
887     // Adding another extension here? Remember to update
888     // of elements in the make() at the top of the funct
889
890     return ret[0:n], nil
891 }
892
893 var (
894     oidSHA1WithRSA = []int{1, 2, 840, 113549, 1, 1, 5}
895     oidRSA         = []int{1, 2, 840, 113549, 1, 1, 1}
896 )
897
898 func subjectBytes(cert *Certificate) ([]byte, error) {
899     if len(cert.RawSubject) > 0 {
900         return cert.RawSubject, nil
901     }
902
903     return asn1.Marshal(cert.Subject.ToRDNSSequence())
904 }
905
906 // CreateCertificate creates a new certificate based on a te
907 // following members of template are used: SerialNumber, Sub
908 // NotAfter, KeyUsage, BasicConstraintsValid, IsCA, MaxPathL
909 // DNSNames, PermittedDNSDomainsCritical, PermittedDNSDomain
910 //
911 // The certificate is signed by parent. If parent is equal t
912 // certificate is self-signed. The parameter pub is the publ
913 // signee and priv is the private key of the signer.
914 //
915 // The returned slice is the certificate in DER encoding.
916 //
917 // The only supported key type is RSA (*rsa.PublicKey for pu
918 // for priv).
919 func CreateCertificate(rand io.Reader, template, parent *Cer
920     rsaPub, ok := pub.(*rsa.PublicKey)
921     if !ok {
922         return nil, errors.New("x509: non-RSA public
923     }
924
925     rsaPriv, ok := priv.(*rsa.PrivateKey)
926     if !ok {
927         return nil, errors.New("x509: non-RSA privat
928     }
929

```

```

930     asn1PublicKey, err := asn1.Marshal(rsaPublicKey{
931         N: rsaPub.N,
932         E: rsaPub.E,
933     })
934     if err != nil {
935         return
936     }
937
938     if len(parent.SubjectKeyId) > 0 {
939         template.AuthorityKeyId = parent.SubjectKeyI
940     }
941
942     extensions, err := buildExtensions(template)
943     if err != nil {
944         return
945     }
946
947     asn1Issuer, err := subjectBytes(parent)
948     if err != nil {
949         return
950     }
951
952     asn1Subject, err := subjectBytes(template)
953     if err != nil {
954         return
955     }
956
957     encodedPublicKey := asn1.BitString{BitLength: len(as
958     c := tbsCertificate{
959         Version:          2,
960         SerialNumber:     template.SerialNumber,
961         SignatureAlgorithm: pkix.AlgorithmIdentifier
962         Issuer:            asn1.RawValue{FullBytes:
963         Validity:         validity{template.NotBef
964         Subject:          asn1.RawValue{FullBytes:
965         PublicKey:        publicKeyInfo{nil, pkix.
966         Extensions:      extensions,
967     }
968
969     tbsCertContents, err := asn1.Marshal(c)
970     if err != nil {
971         return
972     }
973
974     c.Raw = tbsCertContents
975
976     h := sha1.New()
977     h.Write(tbsCertContents)
978     digest := h.Sum(nil)
979

```

```

980     signature, err := rsa.SignPKCS1v15(rand, rsaPriv, cr
981     if err != nil {
982         return
983     }
984
985     cert, err = asn1.Marshal(certificate{
986         nil,
987         c,
988         pkix.AlgorithmIdentifier{Algorithm: oidSHA1w
989         asn1.BitString{Bytes: signature, BitLength:
990     })
991     return
992 }
993
994 // pemCRLPrefix is the magic string that indicates that we h
995 // CRL.
996 var pemCRLPrefix = []byte("-----BEGIN X509 CRL")
997
998 // pemType is the type of a PEM encoded CRL.
999 var pemType = "X509 CRL"
1000
1001 // ParseCRL parses a CRL from the given bytes. It's often th
1002 // encoded CRLs will appear where they should be DER encoded
1003 // will transparently handle PEM encoding as long as there i
1004 // garbage.
1005 func ParseCRL(crlBytes []byte) (certList *pkix.CertificateLi
1006     if bytes.HasPrefix(crlBytes, pemCRLPrefix) {
1007         block, _ := pem.Decode(crlBytes)
1008         if block != nil && block.Type == pemType {
1009             crlBytes = block.Bytes
1010         }
1011     }
1012     return ParseDERCRL(crlBytes)
1013 }
1014
1015 // ParseDERCRL parses a DER encoded CRL from the given bytes
1016 func ParseDERCRL(derBytes []byte) (certList *pkix.Certificat
1017     certList = new(pkix.CertificateList)
1018     _, err = asn1.Unmarshal(derBytes, certList)
1019     if err != nil {
1020         certList = nil
1021     }
1022     return
1023 }
1024
1025 // CreateCRL returns a DER encoded CRL, signed by this Certi
1026 // contains the given list of revoked certificates.
1027 //
1028 // The only supported key type is RSA (*rsa.PrivateKey for p

```

```

1029 func (c *Certificate) CreateCRL(rand io.Reader, priv interfa
1030     rsaPriv, ok := priv.(*rsa.PrivateKey)
1031     if !ok {
1032         return nil, errors.New("x509: non-RSA privat
1033     }
1034     tbsCertList := pkix.TBSCertificateList{
1035         Version: 2,
1036         Signature: pkix.AlgorithmIdentifier{
1037             Algorithm: oidSignatureSHA1WithRSA,
1038         },
1039         Issuer:          c.Subject.ToRDNSSequence
1040         ThisUpdate:     now,
1041         NextUpdate:     expiry,
1042         RevokedCertificates: revokedCerts,
1043     }
1044
1045     tbsCertListContents, err := asn1.Marshal(tbsCertList
1046     if err != nil {
1047         return
1048     }
1049
1050     h := sha1.New()
1051     h.Write(tbsCertListContents)
1052     digest := h.Sum(nil)
1053
1054     signature, err := rsa.SignPKCS1v15(rand, rsaPriv, cr
1055     if err != nil {
1056         return
1057     }
1058
1059     return asn1.Marshal(pkix.CertificateList{
1060         TBSCertList: tbsCertList,
1061         SignatureAlgorithm: pkix.AlgorithmIdentifier
1062             Algorithm: oidSignatureSHA1WithRSA,
1063         },
1064         SignatureValue: asn1.BitString{Bytes: signat
1065     })
1066 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/crypto/x509/pkix/pkix.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package pkix contains shared, low level structures used f
6 // and serialization of X.509 certificates, CRL and OCSP.
7 package pkix
8
9 import (
10     "encoding/asn1"
11     "math/big"
12     "time"
13 )
14
15 // AlgorithmIdentifier represents the ASN.1 structure of the
16 // 5280, section 4.1.1.2.
17 type AlgorithmIdentifier struct {
18     Algorithm  asn1.ObjectIdentifier
19     Parameters asn1.RawValue `asn1:"optional"`
20 }
21
22 type RDNSequence []RelativeDistinguishedNameSET
23
24 type RelativeDistinguishedNameSET []AttributeTypeAndValue
25
26 // AttributeTypeAndValue mirrors the ASN.1 structure of the
27 // http://tools.ietf.org/html/rfc5280#section-4.1.2.4
28 type AttributeTypeAndValue struct {
29     Type  asn1.ObjectIdentifier
30     Value interface{}
31 }
32
33 // Extension represents the ASN.1 structure of the same name
34 // 5280, section 4.2.
35 type Extension struct {
36     Id          asn1.ObjectIdentifier
37     Critical    bool `asn1:"optional"`
38     Value       []byte
39 }
40
41 // Name represents an X.509 distinguished name. This only in
```

```

42 // elements of a DN. Additional elements in the name are ig
43 type Name struct {
44     Country, Organization, OrganizationalUnit []string
45     Locality, Province                        []string
46     StreetAddress, PostalCode                []string
47     SerialNumber, CommonName                 string
48
49     Names []AttributeTypeAndValue
50 }
51
52 func (n *Name) FillFromRDNSequence(rdns *RDNSequence) {
53     for _, rdn := range *rdns {
54         if len(rdn) == 0 {
55             continue
56         }
57         atv := rdn[0]
58         n.Names = append(n.Names, atv)
59         value, ok := atv.Value.(string)
60         if !ok {
61             continue
62         }
63
64         t := atv.Type
65         if len(t) == 4 && t[0] == 2 && t[1] == 5 &&
66             switch t[3] {
67             case 3:
68                 n.CommonName = value
69             case 5:
70                 n.SerialNumber = value
71             case 6:
72                 n.Country = append(n.Country
73             case 7:
74                 n.Locality = append(n.Locali
75             case 8:
76                 n.Province = append(n.Provin
77             case 9:
78                 n.StreetAddress = append(n.S
79             case 10:
80                 n.Organization = append(n.Or
81             case 11:
82                 n.OrganizationalUnit = appen
83             case 17:
84                 n.PostalCode = append(n.Post
85             }
86         }
87     }
88 }
89
90 var (
91     oidCountry = []int{2, 5, 4, 6}

```

```

92         oidOrganization          = []int{2, 5, 4, 10}
93         oidOrganizationalUnit    = []int{2, 5, 4, 11}
94         oidCommonName           = []int{2, 5, 4, 3}
95         oidSerialNumber         = []int{2, 5, 4, 5}
96         oidLocality             = []int{2, 5, 4, 7}
97         oidProvince            = []int{2, 5, 4, 8}
98         oidStreetAddress        = []int{2, 5, 4, 9}
99         oidPostalCode          = []int{2, 5, 4, 17}
100    )
101
102    // appendRDNs appends a relativeDistinguishedNameSET to the
103    // and returns the new value. The relativeDistinguishedNames
104    // attributeTypeAndValue for each of the given values. See R
105    // search for AttributeTypeAndValue.
106    func appendRDNs(in RDNSequence, values []string, oid asn1.ObjectIdentifier) RDNSequence {
107        if len(values) == 0 {
108            return in
109        }
110
111        s := make([]AttributeTypeAndValue, len(values))
112        for i, value := range values {
113            s[i].Type = oid
114            s[i].Value = value
115        }
116
117        return append(in, s)
118    }
119
120    func (n Name) ToRDNSequence() (ret RDNSequence) {
121        ret = appendRDNs(ret, n.Country, oidCountry)
122        ret = appendRDNs(ret, n.Organization, oidOrganization)
123        ret = appendRDNs(ret, n.OrganizationalUnit, oidOrganizationalUnit)
124        ret = appendRDNs(ret, n.Locality, oidLocality)
125        ret = appendRDNs(ret, n.Province, oidProvince)
126        ret = appendRDNs(ret, n.StreetAddress, oidStreetAddress)
127        ret = appendRDNs(ret, n.PostalCode, oidPostalCode)
128        if len(n.CommonName) > 0 {
129            ret = appendRDNs(ret, []string{n.CommonName})
130        }
131        if len(n.SerialNumber) > 0 {
132            ret = appendRDNs(ret, []string{n.SerialNumber})
133        }
134
135        return ret
136    }
137
138    // CertificateList represents the ASN.1 structure of the same
139    // 5280, section 5.1. Use Certificate.CheckCRLSignature to verify
140    // signature.

```

```

141 type CertificateList struct {
142     TBSCertList      TBSCertificateList
143     SignatureAlgorithm AlgorithmIdentifier
144     SignatureValue   asn1.BitString
145 }
146
147 // HasExpired returns true iff now is past the expiry time o
148 func (certList *CertificateList) HasExpired(now time.Time) b
149     return now.After(certList.TBSCertList.NextUpdate)
150 }
151
152 // TBSCertificateList represents the ASN.1 structure of the
153 // 5280, section 5.1.
154 type TBSCertificateList struct {
155     Raw          asn1.RawContent
156     Version      int `asn1:"optional,default:2"`
157     Signature    AlgorithmIdentifier
158     Issuer       RDNSSequence
159     ThisUpdate   time.Time
160     NextUpdate   time.Time
161     RevokedCertificates []RevokedCertificate `asn1:"opti
162     Extensions    []Extension          `asn1:"tag:
163 }
164
165 // RevokedCertificate represents the ASN.1 structure of the
166 // 5280, section 5.1.
167 type RevokedCertificate struct {
168     SerialNumber *big.Int
169     RevocationTime time.Time
170     Extensions    []Extension `asn1:"optional"`
171 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/database/sql/convert.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Type conversions for Scan.
6
7 package sql
8
9 import (
10     "database/sql/driver"
11     "errors"
12     "fmt"
13     "reflect"
14     "strconv"
15 )
16
17 // subsetTypeArgs takes a slice of arguments from callers of
18 // package and converts them into a slice of the driver pack
19 // "subset types".
20 func subsetTypeArgs(args []interface{}) ([]driver.Value, error) {
21     out := make([]driver.Value, len(args))
22     for n, arg := range args {
23         var err error
24         out[n], err = driver.DefaultParameterConvert
25         if err != nil {
26             return nil, fmt.Errorf("sql: convert
27         }
28     }
29     return out, nil
30 }
31
32 // convertAssign copies to dest the value in src, converting
33 // An error is returned if the copy would result in loss of
34 // dest should be a pointer type.
35 func convertAssign(dest, src interface{}) error {
36     // Common cases, without reflect. Fall through.
37     switch s := src.(type) {
38     case string:
39         switch d := dest.(type) {
40         case *string:
41             *d = s
```

```

42         return nil
43     case *[]byte:
44         *d = []byte(s)
45         return nil
46     }
47     case []byte:
48         switch d := dest.(type) {
49         case *string:
50             *d = string(s)
51             return nil
52         case *interface{}:
53             bcopy := make([]byte, len(s))
54             copy(bcopy, s)
55             *d = bcopy
56             return nil
57         case *[]byte:
58             *d = s
59             return nil
60         }
61     case nil:
62         switch d := dest.(type) {
63         case *[]byte:
64             *d = nil
65             return nil
66         }
67     }
68
69     var sv reflect.Value
70
71     switch d := dest.(type) {
72     case *string:
73         sv = reflect.ValueOf(src)
74         switch sv.Kind() {
75         case reflect.Bool,
76             reflect.Int, reflect.Int8, reflect.I
77             reflect.Uint, reflect.Uint8, reflect
78             reflect.Float32, reflect.Float64:
79             *d = fmt.Sprintf("%v", src)
80             return nil
81         }
82     case *bool:
83         bv, err := driver.Bool.ConvertValue(src)
84         if err == nil {
85             *d = bv.(bool)
86         }
87         return err
88     case *interface{}:
89         *d = src
90         return nil
91     }

```

```

92
93     if scanner, ok := dest.(Scanner); ok {
94         return scanner.Scan(src)
95     }
96
97     dpv := reflect.ValueOf(dest)
98     if dpv.Kind() != reflect.Ptr {
99         return errors.New("destination not a pointer")
100    }
101
102    if !sv.IsValid() {
103        sv = reflect.ValueOf(src)
104    }
105
106    dv := reflect.Indirect(dpv)
107    if dv.Kind() == sv.Kind() {
108        dv.Set(sv)
109        return nil
110    }
111
112    switch dv.Kind() {
113    case reflect.Ptr:
114        if src == nil {
115            dv.Set(reflect.Zero(dv.Type()))
116            return nil
117        } else {
118            dv.Set(reflect.New(dv.Type().Elem()))
119            return convertAssign(dv.Interface(),
120        }
121    case reflect.Int, reflect.Int8, reflect.Int16, refle
122        s := asString(src)
123        i64, err := strconv.ParseInt(s, 10, dv.Type(
124        if err != nil {
125            return fmt.Errorf("converting string
126        }
127        dv.SetInt(i64)
128        return nil
129    case reflect.Uint, reflect.Uint8, reflect.Uint16, re
130        s := asString(src)
131        u64, err := strconv.ParseUint(s, 10, dv.Type
132        if err != nil {
133            return fmt.Errorf("converting string
134        }
135        dv.SetUint(u64)
136        return nil
137    case reflect.Float32, reflect.Float64:
138        s := asString(src)
139        f64, err := strconv.ParseFloat(s, dv.Type().
140        if err != nil {

```

```
141             return fmt.Errorf("converting string
142         }
143         dv.SetFloat(f64)
144         return nil
145     }
146
147     return fmt.Errorf("unsupported driver -> Scan pair:
148 }
149
150 func asString(src interface{}) string {
151     switch v := src.(type) {
152     case string:
153         return v
154     case []byte:
155         return string(v)
156     }
157     return fmt.Sprintf("%v", src)
158 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/database/sql/sql.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package sql provides a generic interface around SQL (or S
6 // databases.
7 package sql
8
9 import (
10     "database/sql/driver"
11     "errors"
12     "fmt"
13     "io"
14     "sync"
15 )
16
17 var drivers = make(map[string]driver.Driver)
18
19 // Register makes a database driver available by the provide
20 // If Register is called twice with the same name or if driv
21 // it panics.
22 func Register(name string, driver driver.Driver) {
23     if driver == nil {
24         panic("sql: Register driver is nil")
25     }
26     if _, dup := drivers[name]; dup {
27         panic("sql: Register called twice for driver")
28     }
29     drivers[name] = driver
30 }
31
32 // RawBytes is a byte slice that holds a reference to memory
33 // the database itself. After a Scan into a RawBytes, the sl
34 // valid until the next call to Next, Scan, or Close.
35 type RawBytes []byte
36
37 // NullString represents a string that may be null.
38 // NullString implements the Scanner interface so
39 // it can be used as a scan destination:
40 //
41 // var s NullString
42 // err := db.QueryRow("SELECT name FROM foo WHERE id=?", id
43 // ...
44 // if s.Valid {
```

```

45 //      // use s.String
46 // } else {
47 //      // NULL value
48 // }
49 //
50 type NullString struct {
51     String string
52     Valid bool // Valid is true if String is not NULL
53 }
54
55 // Scan implements the Scanner interface.
56 func (ns *NullString) Scan(value interface{}) error {
57     if value == nil {
58         ns.String, ns.Valid = "", false
59         return nil
60     }
61     ns.Valid = true
62     return convertAssign(&ns.String, value)
63 }
64
65 // Value implements the driver Valuer interface.
66 func (ns NullString) Value() (driver.Value, error) {
67     if !ns.Valid {
68         return nil, nil
69     }
70     return ns.String, nil
71 }
72
73 // NullInt64 represents an int64 that may be null.
74 // NullInt64 implements the Scanner interface so
75 // it can be used as a scan destination, similar to NullStri
76 type NullInt64 struct {
77     Int64 int64
78     Valid bool // Valid is true if Int64 is not NULL
79 }
80
81 // Scan implements the Scanner interface.
82 func (n *NullInt64) Scan(value interface{}) error {
83     if value == nil {
84         n.Int64, n.Valid = 0, false
85         return nil
86     }
87     n.Valid = true
88     return convertAssign(&n.Int64, value)
89 }
90
91 // Value implements the driver Valuer interface.
92 func (n NullInt64) Value() (driver.Value, error) {
93     if !n.Valid {
94         return nil, nil

```

```

95         }
96         return n.Int64, nil
97     }
98
99     // NullFloat64 represents a float64 that may be null.
100    // NullFloat64 implements the Scanner interface so
101    // it can be used as a scan destination, similar to NullStri
102    type NullFloat64 struct {
103        Float64 float64
104        Valid   bool // Valid is true if Float64 is not NULL
105    }
106
107    // Scan implements the Scanner interface.
108    func (n *NullFloat64) Scan(value interface{}) error {
109        if value == nil {
110            n.Float64, n.Valid = 0, false
111            return nil
112        }
113        n.Valid = true
114        return convertAssign(&n.Float64, value)
115    }
116
117    // Value implements the driver Valuer interface.
118    func (n NullFloat64) Value() (driver.Value, error) {
119        if !n.Valid {
120            return nil, nil
121        }
122        return n.Float64, nil
123    }
124
125    // NullBool represents a bool that may be null.
126    // NullBool implements the Scanner interface so
127    // it can be used as a scan destination, similar to NullStri
128    type NullBool struct {
129        Bool   bool
130        Valid bool // Valid is true if Bool is not NULL
131    }
132
133    // Scan implements the Scanner interface.
134    func (n *NullBool) Scan(value interface{}) error {
135        if value == nil {
136            n.Bool, n.Valid = false, false
137            return nil
138        }
139        n.Valid = true
140        return convertAssign(&n.Bool, value)
141    }
142
143    // Value implements the driver Valuer interface.

```

```

144 func (n NullBool) Value() (driver.Value, error) {
145     if !n.Valid {
146         return nil, nil
147     }
148     return n.Bool, nil
149 }
150
151 // Scanner is an interface used by Scan.
152 type Scanner interface {
153     // Scan assigns a value from a database driver.
154     //
155     // The src value will be of one of the following res
156     // set of types:
157     //
158     //     int64
159     //     float64
160     //     bool
161     //     []byte
162     //     string
163     //     time.Time
164     //     nil - for NULL values
165     //
166     // An error should be returned if the value can not
167     // without loss of information.
168     Scan(src interface{}) error
169 }
170
171 // ErrNoRows is returned by Scan when QueryRow doesn't retur
172 // row. In such a case, QueryRow returns a placeholder *Row
173 // defers this error until a Scan.
174 var ErrNoRows = errors.New("sql: no rows in result set")
175
176 // DB is a database handle. It's safe for concurrent use by
177 // goroutines.
178 //
179 // If the underlying database driver has the concept of a co
180 // and per-connection session state, the sql package manages
181 // and freeing connections automatically, including maintain
182 // pool of idle connections. If observing session state is r
183 // either do not share a *DB between multiple concurrent gor
184 // create and observe all state only within a transaction. 0
185 // DB.Open is called, the returned Tx is bound to a single i
186 // connection. Once Tx.Commit or Tx.Rollback is called, that
187 // connection is returned to DB's idle connection pool.
188 type DB struct {
189     driver driver.Driver
190     dsn     string
191
192     mu     sync.Mutex // protects freeConn and closed

```

```

193         freeConn []driver.Conn
194         closed    bool
195     }
196
197     // Open opens a database specified by its database driver name
198     // driver-specific data source name, usually consisting of a
199     // database name and connection information.
200     //
201     // Most users will open a database via a driver-specific connection
202     // helper function that returns a *DB.
203     func Open(driverName, dataSourceName string) (*DB, error) {
204         driver, ok := drivers[driverName]
205         if !ok {
206             return nil, fmt.Errorf("sql: unknown driver
207
208         }
209         return &DB{driver: driver, dsn: dataSourceName}, nil
210     }
211
212     // Close closes the database, releasing any open resources.
213     func (db *DB) Close() error {
214         db.mu.Lock()
215         defer db.mu.Unlock()
216         var err error
217         for _, c := range db.freeConn {
218             err1 := c.Close()
219             if err1 != nil {
220                 err = err1
221             }
222         }
223         db.freeConn = nil
224         db.closed = true
225         return err
226     }
227
228     func (db *DB) maxIdleConns() int {
229         const defaultMaxIdleConns = 2
230         // TODO(bradfitz): ask driver, if supported, for its
231         // TODO(bradfitz): let users override?
232         return defaultMaxIdleConns
233     }
234
235     // conn returns a newly-opened or cached driver.Conn
236     func (db *DB) conn() (driver.Conn, error) {
237         db.mu.Lock()
238         if db.closed {
239             db.mu.Unlock()
240             return nil, errors.New("sql: database is closed")
241         }
242         if n := len(db.freeConn); n > 0 {

```

```

243         db.freeConn = db.freeConn[:n-1]
244         db.mu.Unlock()
245         return conn, nil
246     }
247     db.mu.Unlock()
248     return db.driver.Open(db.dsn)
249 }
250
251 func (db *DB) connIfFree(wanted driver.Conn) (conn driver.Co
252     db.mu.Lock()
253     defer db.mu.Unlock()
254     for i, conn := range db.freeConn {
255         if conn != wanted {
256             continue
257         }
258         db.freeConn[i] = db.freeConn[len(db.freeConn)
259         db.freeConn = db.freeConn[:len(db.freeConn)-
260         return wanted, true
261     }
262     return nil, false
263 }
264
265 // putConnHook is a hook for testing.
266 var putConnHook func(*DB, driver.Conn)
267
268 // putConn adds a connection to the db's free pool.
269 // err is optionally the last error that occurred on this con
270 func (db *DB) putConn(c driver.Conn, err error) {
271     if err == driver.ErrBadConn {
272         // Don't reuse bad connections.
273         return
274     }
275     db.mu.Lock()
276     if putConnHook != nil {
277         putConnHook(db, c)
278     }
279     if n := len(db.freeConn); !db.closed && n < db.maxId
280         db.freeConn = append(db.freeConn, c)
281         db.mu.Unlock()
282         return
283     }
284     // TODO: check to see if we need this Conn for any p
285     // statements which are still active?
286     db.mu.Unlock()
287     c.Close()
288 }
289
290 // Prepare creates a prepared statement for later execution.
291 func (db *DB) Prepare(query string) (*Stmt, error) {

```

```

292     var stmt *Stmt
293     var err error
294     for i := 0; i < 10; i++ {
295         stmt, err = db.prepare(query)
296         if err != driver.ErrBadConn {
297             break
298         }
299     }
300     return stmt, err
301 }
302
303 func (db *DB) prepare(query string) (stmt *Stmt, err error) {
304     // TODO: check if db.driver supports an optional
305     // driver.Preparer interface and call that instead,
306     // otherwise we make a prepared statement that's bou
307     // to a connection, and to execute this prepared sta
308     // we either need to use this connection (if it's fr
309     // get a new connection + re-prepare + execute on th
310     ci, err := db.conn()
311     if err != nil {
312         return nil, err
313     }
314     defer db.putConn(ci, err)
315     si, err := ci.Prepare(query)
316     if err != nil {
317         return nil, err
318     }
319     stmt = &Stmt{
320         db:    db,
321         query: query,
322         css:   []connStmt{{ci, si}},
323     }
324     return stmt, nil
325 }
326
327 // Exec executes a query without returning any rows.
328 func (db *DB) Exec(query string, args ...interface{}) (Resul
329     sargs, err := subsetTypeArgs(args)
330     if err != nil {
331         return nil, err
332     }
333     var res Result
334     for i := 0; i < 10; i++ {
335         res, err = db.exec(query, sargs)
336         if err != driver.ErrBadConn {
337             break
338         }
339     }
340     return res, err

```

```

341 }
342
343 func (db *DB) exec(query string, sargs []driver.Value) (res
344     ci, err := db.conn()
345     if err != nil {
346         return nil, err
347     }
348     defer db.putConn(ci, err)
349
350     if excec, ok := ci.(driver.Execer); ok {
351         resi, err := excec.Exec(query, sargs)
352         if err != driver.ErrSkip {
353             if err != nil {
354                 return nil, err
355             }
356             return result{resi}, nil
357         }
358     }
359
360     sti, err := ci.Prepare(query)
361     if err != nil {
362         return nil, err
363     }
364     defer sti.Close()
365
366     resi, err := sti.Exec(sargs)
367     if err != nil {
368         return nil, err
369     }
370     return result{resi}, nil
371 }
372
373 // Query executes a query that returns rows, typically a SEL
374 func (db *DB) Query(query string, args ...interface{}) (*Row
375     stmt, err := db.Prepare(query)
376     if err != nil {
377         return nil, err
378     }
379     rows, err := stmt.Query(args...)
380     if err != nil {
381         stmt.Close()
382         return nil, err
383     }
384     rows.closeStmt = stmt
385     return rows, nil
386 }
387
388 // QueryRow executes a query that is expected to return at m
389 // QueryRow always return a non-nil value. Errors are deferr
390 // Row's Scan method is called.

```

```

391 func (db *DB) QueryRow(query string, args ...interface{}) *R
392     rows, err := db.Query(query, args...)
393     return &Row{rows: rows, err: err}
394 }
395
396 // Begin starts a transaction. The isolation level is depend
397 // the driver.
398 func (db *DB) Begin() (*Tx, error) {
399     var tx *Tx
400     var err error
401     for i := 0; i < 10; i++ {
402         tx, err = db.begin()
403         if err != driver.ErrBadConn {
404             break
405         }
406     }
407     return tx, err
408 }
409
410 func (db *DB) begin() (tx *Tx, err error) {
411     ci, err := db.conn()
412     if err != nil {
413         return nil, err
414     }
415     txi, err := ci.Begin()
416     if err != nil {
417         db.putConn(ci, err)
418         return nil, fmt.Errorf("sql: failed to Begin
419     }
420     return &Tx{
421         db: db,
422         ci: ci,
423         txi: txi,
424     }, nil
425 }
426
427 // Driver returns the database's underlying driver.
428 func (db *DB) Driver() driver.Driver {
429     return db.driver
430 }
431
432 // Tx is an in-progress database transaction.
433 //
434 // A transaction must end with a call to Commit or Rollback.
435 //
436 // After a call to Commit or Rollback, all operations on the
437 // transaction fail with ErrTxDone.
438 type Tx struct {
439     db *DB

```

```

440
441     // ci is owned exclusively until Commit or Rollback,
442     // it's returned with putConn.
443     ci driver.Conn
444     txi driver.Tx
445
446     // cimu is held while somebody is using ci (between
447     // and releaseConn)
448     cimu sync.Mutex
449
450     // done transitions from false to true exactly once,
451     // or Rollback. once done, all operations fail with
452     // ErrTxDone.
453     done bool
454 }
455
456 var ErrTxDone = errors.New("sql: Transaction has already bee
457
458 func (tx *Tx) close() {
459     if tx.done {
460         panic("double close") // internal error
461     }
462     tx.done = true
463     tx.db.putConn(tx.ci, nil)
464     tx.ci = nil
465     tx.txi = nil
466 }
467
468 func (tx *Tx) grabConn() (driver.Conn, error) {
469     if tx.done {
470         return nil, ErrTxDone
471     }
472     tx.cimu.Lock()
473     return tx.ci, nil
474 }
475
476 func (tx *Tx) releaseConn() {
477     tx.cimu.Unlock()
478 }
479
480 // Commit commits the transaction.
481 func (tx *Tx) Commit() error {
482     if tx.done {
483         return ErrTxDone
484     }
485     defer tx.close()
486     return tx.txi.Commit()
487 }
488

```

```

489 // Rollback aborts the transaction.
490 func (tx *Tx) Rollback() error {
491     if tx.done {
492         return ErrTxDone
493     }
494     defer tx.close()
495     return tx.txi.Rollback()
496 }
497
498 // Prepare creates a prepared statement for use within a tra
499 //
500 // The returned statement operates within the transaction an
501 // be used once the transaction has been committed or rolled
502 //
503 // To use an existing prepared statement on this transaction
504 func (tx *Tx) Prepare(query string) (*Stmt, error) {
505     // TODO(bradfitz): We could be more efficient here a
506     // provide a method to take an existing Stmt (create
507     // perhaps a different Conn), and re-create it on th
508     // necessary. Or, better: keep a map in DB of query
509     // Stmts, and have Stmt.Execute do the right thing a
510     // re-prepare if the Conn in use doesn't have that p
511     // statement. But we'll want to avoid caching the s
512     // in the case where we only call conn.Prepare impli
513     // (such as in db.Exec or tx.Exec), but the caller p
514     // can't be holding a reference to the returned stat
515     // Perhaps just looking at the reference count (by n
516     // Stmt.Close) would be enough. We might also want a
517     // on Stmt to drop the reference count.
518     ci, err := tx.grabConn()
519     if err != nil {
520         return nil, err
521     }
522     defer tx.releaseConn()
523
524     si, err := ci.Prepare(query)
525     if err != nil {
526         return nil, err
527     }
528
529     stmt := &Stmt{
530         db:    tx.db,
531         tx:    tx,
532         txsi:  si,
533         query: query,
534     }
535     return stmt, nil
536 }
537
538 // Stmt returns a transaction-specific prepared statement fr

```

```

539 // an existing statement.
540 //
541 // Example:
542 // updateMoney, err := db.Prepare("UPDATE balance SET money
543 // ...
544 // tx, err := db.Begin()
545 // ...
546 // res, err := tx.Stmt(updateMoney).Exec(123.45, 98293203)
547 func (tx *Tx) Stmt(stmt *Stmt) *Stmt {
548     // TODO(bradfitz): optimize this. Currently this re-
549     // each time. This is fine for now to illustrate th
550     // we should really cache already-prepared statement
551     // per-Conn. See also the big comment in Tx.Prepare.
552
553     if tx.db != stmt.db {
554         return &Stmt{stickyErr: errors.New("sql: Tx.
555     }
556     ci, err := tx.grabConn()
557     if err != nil {
558         return &Stmt{stickyErr: err}
559     }
560     defer tx.releaseConn()
561     si, err := ci.Prepare(stmt.query)
562     return &Stmt{
563         db:      tx.db,
564         tx:      tx,
565         txsi:    si,
566         query:   stmt.query,
567         stickyErr: err,
568     }
569 }
570
571 // Exec executes a query that doesn't return rows.
572 // For example: an INSERT and UPDATE.
573 func (tx *Tx) Exec(query string, args ...interface{}) (Resul
574     ci, err := tx.grabConn()
575     if err != nil {
576         return nil, err
577     }
578     defer tx.releaseConn()
579
580     sargs, err := subsetTypeArgs(args)
581     if err != nil {
582         return nil, err
583     }
584
585     if execer, ok := ci.(driver.Execer); ok {
586         resi, err := execer.Exec(query, sargs)
587         if err == nil {

```

```

588             return result{resi}, nil
589         }
590         if err != driver.ErrSkip {
591             return nil, err
592         }
593     }
594
595     sti, err := ci.Prepare(query)
596     if err != nil {
597         return nil, err
598     }
599     defer sti.Close()
600
601     resi, err := sti.Exec(sargs)
602     if err != nil {
603         return nil, err
604     }
605     return result{resi}, nil
606 }
607
608 // Query executes a query that returns rows, typically a SEL
609 func (tx *Tx) Query(query string, args ...interface{}) (*Row
610     if tx.done {
611         return nil, ErrTxDone
612     }
613     stmt, err := tx.Prepare(query)
614     if err != nil {
615         return nil, err
616     }
617     rows, err := stmt.Query(args...)
618     if err != nil {
619         stmt.Close()
620         return nil, err
621     }
622     rows.closeStmt = stmt
623     return rows, err
624 }
625
626 // QueryRow executes a query that is expected to return at m
627 // QueryRow always return a non-nil value. Errors are deferr
628 // Row's Scan method is called.
629 func (tx *Tx) QueryRow(query string, args ...interface{}) *R
630     rows, err := tx.Query(query, args...)
631     return &Row{rows: rows, err: err}
632 }
633
634 // connStmt is a prepared statement on a particular connecti
635 type connStmt struct {
636     ci driver.Conn

```

```

637         si driver.Stmt
638     }
639
640 // Stmt is a prepared statement. Stmt is safe for concurrent
641 type Stmt struct {
642     // Immutable:
643     db      *DB      // where we came from
644     query   string  // that created the Stmt
645     stickyErr error // if non-nil, this error is return
646
647     // If in a transaction, else both nil:
648     tx      *Tx
649     txsi driver.Stmt
650
651     mu      sync.Mutex // protects the rest of the fields
652     closed bool
653
654     // css is a list of underlying driver statement inte
655     // that are valid on particular connections. This i
656     // used if tx == nil and one is found that has idle
657     // connections. If tx != nil, txsi is always used.
658     css []connStmt
659 }
660
661 // Exec executes a prepared statement with the given argumen
662 // returns a Result summarizing the effect of the statement.
663 func (s *Stmt) Exec(args ...interface{}) (Result, error) {
664     _, releaseConn, si, err := s.connStmt()
665     if err != nil {
666         return nil, err
667     }
668     defer releaseConn(nil)
669
670     // -1 means the driver doesn't know how to count the
671     // placeholders, so we won't sanity check input here
672     // driver deal with errors.
673     if want := si.NumInput(); want != -1 && len(args) !=
674         want {
675         return nil, fmt.Errorf("sql: expected %d arg
676     }
677
678     sargs := make([]driver.Value, len(args))
679
680     // Convert args to subset types.
681     if cc, ok := si.(driver.ColumnConverter); ok {
682         for n, arg := range args {
683             // First, see if the value itself kn
684             // itself to a driver type. For exa
685             // struct changing into a string or
686             if svi, ok := arg.(driver.Valuer); o
687                 sv, err := svi.Value()

```

```

687         if err != nil {
688             return nil, fmt.Erro
689         }
690         if !driver.IsValue(sv) {
691             return nil, fmt.Erro
692         }
693         arg = sv
694     }
695
696     // Second, ask the column to sanity
697     // example, drivers might use this t
698     // an int64 values being inserted in
699     // integer field is in range (before
700     // truncated), or that a nil can't g
701     // column before going across the ne
702     // same error.
703     sargs[n], err = cc.ColumnConverter(n
704     if err != nil {
705         return nil, fmt.Errorf("sql:
706     }
707     if !driver.IsValue(sargs[n]) {
708         return nil, fmt.Errorf("sql:
709         arg, sargs[n])
710     }
711     }
712     } else {
713         for n, arg := range args {
714             sargs[n], err = driver.DefaultParame
715             if err != nil {
716                 return nil, fmt.Errorf("sql:
717             }
718         }
719     }
720
721     resi, err := si.Exec(sargs)
722     if err != nil {
723         return nil, err
724     }
725     return result{resi}, nil
726 }
727
728 // connStmt returns a free driver connection on which to exe
729 // statement, a function to call to release the connection,
730 // statement bound to that connection.
731 func (s *Stmt) connStmt() (ci driver.Conn, releaseConn func(
732     if err = s.stickyErr; err != nil {
733         return
734     }
735     s.mu.Lock()

```

```

736     if s.closed {
737         s.mu.Unlock()
738         err = errors.New("sql: statement is closed")
739         return
740     }
741
742     // In a transaction, we always use the connection th
743     // transaction was created on.
744     if s.tx != nil {
745         s.mu.Unlock()
746         ci, err = s.tx.grabConn() // blocks, waiting
747         if err != nil {
748             return
749         }
750         releaseConn = func(error) { s.tx.releaseConn
751         return ci, releaseConn, s.txsi, nil
752     }
753
754     var cs connStmt
755     match := false
756     for _, v := range s.css {
757         // TODO(bradfitz): lazily clean up entries i
758         // list with dead conns while enumerating
759         if _, match = s.db.connIfFree(v.ci); match {
760             cs = v
761             break
762         }
763     }
764     s.mu.Unlock()
765
766     // Make a new conn if all are busy.
767     // TODO(bradfitz): or wait for one? make configurabl
768     if !match {
769         for i := 0; ; i++ {
770             ci, err := s.db.conn()
771             if err != nil {
772                 return nil, nil, nil, err
773             }
774             si, err := ci.Prepare(s.query)
775             if err == driver.ErrBadConn && i < 1
776                 continue
777             }
778             if err != nil {
779                 return nil, nil, nil, err
780             }
781             s.mu.Lock()
782             cs = connStmt{ci, si}
783             s.css = append(s.css, cs)
784             s.mu.Unlock()

```

```

785             break
786         }
787     }
788
789     conn := cs.ci
790     releaseConn = func(err error) { s.db.putConn(conn, e
791     return conn, releaseConn, cs.si, nil
792 }
793
794 // Query executes a prepared query statement with the given
795 // and returns the query results as a *Rows.
796 func (s *Stmt) Query(args ...interface{}) (*Rows, error) {
797     ci, releaseConn, si, err := s.connStmt()
798     if err != nil {
799         return nil, err
800     }
801
802     // -1 means the driver doesn't know how to count the
803     // placeholders, so we won't sanity check input here
804     // driver deal with errors.
805     if want := si.NumInput(); want != -1 && len(args) !=
806         return nil, fmt.Errorf("sql: statement expect
807     }
808     sargs, err := subsetTypeArgs(args)
809     if err != nil {
810         return nil, err
811     }
812     rowsi, err := si.Query(sargs)
813     if err != nil {
814         releaseConn(err)
815         return nil, err
816     }
817     // Note: ownership of ci passes to the *Rows, to be
818     // with releaseConn.
819     rows := &Rows{
820         db:         s.db,
821         ci:         ci,
822         releaseConn: releaseConn,
823         rowsi:      rowsi,
824     }
825     return rows, nil
826 }
827
828 // QueryRow executes a prepared query statement with the giv
829 // If an error occurs during the execution of the statement,
830 // be returned by a call to Scan on the returned *Row, which
831 // If the query selects no rows, the *Row's Scan will return
832 // Otherwise, the *Row's Scan scans the first selected row a
833 // the rest.
834 //

```

```

835 // Example usage:
836 //
837 // var name string
838 // err := nameByUserIdStmt.QueryRow(id).Scan(&name)
839 func (s *Stmt) QueryRow(args ...interface{}) *Row {
840     rows, err := s.Query(args...)
841     if err != nil {
842         return &Row{err: err}
843     }
844     return &Row{rows: rows}
845 }
846
847 // Close closes the statement.
848 func (s *Stmt) Close() error {
849     if s.stickyErr != nil {
850         return s.stickyErr
851     }
852     s.mu.Lock()
853     defer s.mu.Unlock()
854     if s.closed {
855         return nil
856     }
857     s.closed = true
858
859     if s.tx != nil {
860         s.txsi.Close()
861     } else {
862         for _, v := range s.css {
863             if ci, match := s.db.connIfFree(v.ci) {
864                 v.si.Close()
865                 s.db.putConn(ci, nil)
866             } else {
867                 // TODO(bradfitz): care that
868                 // this statement because th
869                 // connection is in use?
870             }
871         }
872     }
873     return nil
874 }
875
876 // Rows is the result of a query. Its cursor starts before t
877 // of the result set. Use Next to advance through the rows:
878 //
879 //     rows, err := db.Query("SELECT ...")
880 //     ...
881 //     for rows.Next() {
882 //         var id int
883 //         var name string

```

```

884 //          err = rows.Scan(&id, &name)
885 //          ...
886 //      }
887 //      err = rows.Err() // get any error encountered during
888 //      ...
889 type Rows struct {
890     db          *DB
891     ci          driver.Conn // owned; must call putconn
892     releaseConn func(error)
893     rowsi       driver.Rows
894
895     closed      bool
896     lastcols    []driver.Value
897     lasterr     error
898     closeStmt   *Stmt // if non-nil, statement to Close on
899 }
900
901 // Next prepares the next result row for reading with the Sc
902 // It returns true on success, false if there is no next res
903 // Every call to Scan, even the first one, must be preceded
904 // to Next.
905 func (rs *Rows) Next() bool {
906     if rs.closed {
907         return false
908     }
909     if rs.lasterr != nil {
910         return false
911     }
912     if rs.lastcols == nil {
913         rs.lastcols = make([]driver.Value, len(rs.ro
914     }
915     rs.lasterr = rs.rowsi.Next(rs.lastcols)
916     if rs.lasterr == io.EOF {
917         rs.Close()
918     }
919     return rs.lasterr == nil
920 }
921
922 // Err returns the error, if any, that was encountered durin
923 func (rs *Rows) Err() error {
924     if rs.lasterr == io.EOF {
925         return nil
926     }
927     return rs.lasterr
928 }
929
930 // Columns returns the column names.
931 // Columns returns an error if the rows are closed, or if th
932 // are from QueryRow and there was a deferred error.

```

```

933 func (rs *Rows) Columns() ([]string, error) {
934     if rs.closed {
935         return nil, errors.New("sql: Rows are closed")
936     }
937     if rs.rowsi == nil {
938         return nil, errors.New("sql: no Rows available")
939     }
940     return rs.rowsi.Columns(), nil
941 }
942
943 // Scan copies the columns in the current row into the value
944 // at by dest.
945 //
946 // If an argument has type *[]byte, Scan saves in that argument
947 // of the corresponding data. The copy is owned by the caller
948 // and may be modified and held indefinitely. The copy can be avoided
949 // by providing an argument of type *RawBytes instead; see the documentation
950 // for RawBytes for restrictions on its use.
951 //
952 // If an argument has type *interface{}, Scan copies the value
953 // provided by the underlying driver without conversion. If
954 // the argument is of type []byte, a copy is made and the caller owns the
955 // copy.
956 func (rs *Rows) Scan(dest ...interface{}) error {
957     if rs.closed {
958         return errors.New("sql: Rows closed")
959     }
960     if rs.lasterr != nil {
961         return rs.lasterr
962     }
963     if rs.lastcols == nil {
964         return errors.New("sql: Scan called without columns")
965     }
966     if len(dest) != len(rs.lastcols) {
967         return fmt.Errorf("sql: expected %d destination arguments, got %d", len(rs.lastcols), len(dest))
968     }
969     for i, sv := range rs.lastcols {
970         err := convertAssign(dest[i], sv)
971         if err != nil {
972             return fmt.Errorf("sql: Scan error on column %d: %v", i, err)
973         }
974     }
975     for _, dp := range dest {
976         b, ok := dp.(*[]byte)
977         if !ok {
978             continue
979         }
980         if *b == nil {
981             // If the []byte is now nil (for a NULL value),
982             // don't fall through to below which would
983             // turn it into a non-nil 0-length byte slice.

```

```

983             continue
984         }
985         if _, ok = dp.(*RawBytes); ok {
986             continue
987         }
988         clone := make([]byte, len(*b))
989         copy(clone, *b)
990         *b = clone
991     }
992     return nil
993 }
994
995 // Close closes the Rows, preventing further enumeration. If
996 // end is encountered, the Rows are closed automatically. Cl
997 // is idempotent.
998 func (rs *Rows) Close() error {
999     if rs.closed {
1000         return nil
1001     }
1002     rs.closed = true
1003     err := rs.rowsi.Close()
1004     rs.releaseConn(err)
1005     if rs.closeStmt != nil {
1006         rs.closeStmt.Close()
1007     }
1008     return err
1009 }
1010
1011 // Row is the result of calling QueryRow to select a single
1012 type Row struct {
1013     // One of these two will be non-nil:
1014     err error // deferred error for easy chaining
1015     rows *Rows
1016 }
1017
1018 // Scan copies the columns from the matched row into the val
1019 // pointed at by dest. If more than one row matches the que
1020 // Scan uses the first row and discards the rest. If no row
1021 // the query, Scan returns ErrNoRows.
1022 func (r *Row) Scan(dest ...interface{}) error {
1023     if r.err != nil {
1024         return r.err
1025     }
1026
1027     // TODO(bradfitz): for now we need to defensively cl
1028     // []byte that the driver returned (not permitting
1029     // *RawBytes in Rows.Scan), since we're about to clo
1030     // the Rows in our defer, when we return from this f
1031     // the contract with the driver.Next(...) interface

```

```

1032         // can return slices into read-only temporary memory
1033         // only valid until the next Scan/Close. But the TC
1034         // for a lot of drivers, this copy will be unneccessa
1035         // should provide an optional interface for drivers
1036         // implement to say, "don't worry, the []bytes that
1037         // from Next will not be modified again." (for insta
1038         // they were obtained from the network anyway) But f
1039         // don't care.
1040         for _, dp := range dest {
1041             if _, ok := dp.(*RawBytes); ok {
1042                 return errors.New("sql: RawBytes isn
1043             }
1044         }
1045
1046         defer r.rows.Close()
1047         if !r.rows.Next() {
1048             return ErrNoRows
1049         }
1050         err := r.rows.Scan(dest...)
1051         if err != nil {
1052             return err
1053         }
1054
1055         return nil
1056     }
1057
1058     // A Result summarizes an executed SQL command.
1059     type Result interface {
1060         LastInsertId() (int64, error)
1061         RowsAffected() (int64, error)
1062     }
1063
1064     type result struct {
1065         driver.Result
1066     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/database/sql/driver/driver.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package driver defines interfaces to be implemented by da
6 // drivers as used by package sql.
7 //
8 // Most code should use package sql.
9 package driver
10
11 import "errors"
12
13 // A driver Value is a value that drivers must be able to ha
14 // A Value is either nil or an instance of one of these type
15 //
16 // int64
17 // float64
18 // bool
19 // []byte
20 // string [*] everywhere except from Rows.Next.
21 // time.Time
22 type Value interface{}
23
24 // Driver is the interface that must be implemented by a dat
25 // driver.
26 type Driver interface {
27     // Open returns a new connection to the database.
28     // The name is a string in a driver-specific format.
29     //
30     // Open may return a cached connection (one previous
31     // closed), but doing so is unnecessary; the sql pac
32     // maintains a pool of idle connections for efficien
33     //
34     // The returned connection is only used by one gorou
35     // time.
36     Open(name string) (Conn, error)
37 }
38
39 // ErrSkip may be returned by some optional interfaces' meth
40 // indicate at runtime that the fast path is unavailable and
41 // package should continue as if the optional interface was
```

```

42 // implemented. ErrSkip is only supported where explicitly
43 // documented.
44 var ErrSkip = errors.New("driver: skip fast-path; continue a
45
46 // ErrBadConn should be returned by a driver to signal to th
47 // package that a driver.Conn is in a bad state (such as the
48 // having earlier closed the connection) and the sql package
49 // retry on a new connection.
50 //
51 // To prevent duplicate operations, ErrBadConn should NOT be
52 // if there's a possibility that the database server might h
53 // performed the operation. Even if the server sends back an
54 // you shouldn't return ErrBadConn.
55 var ErrBadConn = errors.New("driver: bad connection")
56
57 // Execer is an optional interface that may be implemented b
58 //
59 // If a Conn does not implement Execer, the db package's DB.
60 // first prepare a query, execute the statement, and then cl
61 // statement.
62 //
63 // Exec may return ErrSkip.
64 type Execer interface {
65     Exec(query string, args []Value) (Result, error)
66 }
67
68 // Conn is a connection to a database. It is not used concur
69 // by multiple goroutines.
70 //
71 // Conn is assumed to be stateful.
72 type Conn interface {
73     // Prepare returns a prepared statement, bound to th
74     Prepare(query string) (Stmt, error)
75
76     // Close invalidates and potentially stops any curre
77     // prepared statements and transactions, marking thi
78     // connection as no longer in use.
79     //
80     // Because the sql package maintains a free pool of
81     // connections and only calls Close when there's a s
82     // idle connections, it shouldn't be necessary for d
83     // do their own connection caching.
84     Close() error
85
86     // Begin starts and returns a new transaction.
87     Begin() (Tx, error)
88 }
89
90 // Result is the result of a query execution.
91 type Result interface {

```

```

92         // LastInsertId returns the database's auto-generate
93         // after, for example, an INSERT into a table with p
94         // key.
95         LastInsertId() (int64, error)
96
97         // RowsAffected returns the number of rows affected
98         // query.
99         RowsAffected() (int64, error)
100    }
101
102    // Stmt is a prepared statement. It is bound to a Conn and n
103    // used by multiple goroutines concurrently.
104    type Stmt interface {
105        // Close closes the statement.
106        //
107        // Closing a statement should not interrupt any outs
108        // query created from that statement. That is, the f
109        // order of operations is valid:
110        //
111        // * create a driver statement
112        // * call Query on statement, returning Rows
113        // * close the statement
114        // * read from Rows
115        //
116        // If closing a statement invalidates currently-runn
117        // queries, the final step above will incorrectly fa
118        //
119        // TODO(bradfitz): possibly remove the restriction a
120        // enough driver authors object and find it complica
121        // code too much. The sql package could be smarter a
122        // refcounting the statement and closing it at the a
123        // time.
124        Close() error
125
126        // NumInput returns the number of placeholder parame
127        //
128        // If NumInput returns >= 0, the sql package will sa
129        // argument counts from callers and return errors to
130        // before the statement's Exec or Query methods are
131        //
132        // NumInput may also return -1, if the driver doesn'
133        // its number of placeholders. In that case, the sql
134        // will not sanity check Exec or Query argument coun
135        NumInput() int
136
137        // Exec executes a query that doesn't return rows, s
138        // as an INSERT or UPDATE.
139        Exec(args []Value) (Result, error)
140

```

```

141         // Exec executes a query that may return rows, such
142         // SELECT.
143         Query(args []Value) (Rows, error)
144     }
145
146     // ColumnConverter may be optionally implemented by Stmt if
147     // the statement is aware of its own columns' types and can
148     // convert from any type to a driver Value.
149     type ColumnConverter interface {
150         // ColumnConverter returns a ValueConverter for the
151         // column index. If the type of a specific column i
152         // or shouldn't be handled specially, DefaultValueCo
153         // can be returned.
154         ColumnConverter(idx int) ValueConverter
155     }
156
157     // Rows is an iterator over an executed query's results.
158     type Rows interface {
159         // Columns returns the names of the columns. The num
160         // columns of the result is inferred from the length
161         // slice. If a particular column name isn't known,
162         // string should be returned for that entry.
163         Columns() []string
164
165         // Close closes the rows iterator.
166         Close() error
167
168         // Next is called to populate the next row of data i
169         // the provided slice. The provided slice will be th
170         // size as the Columns() are wide.
171         //
172         // The dest slice may be populated only with
173         // a driver Value type, but excluding string.
174         // All string values must be converted to []byte.
175         //
176         // Next should return io.EOF when there are no more
177         Next(dest []Value) error
178     }
179
180     // Tx is a transaction.
181     type Tx interface {
182         Commit() error
183         Rollback() error
184     }
185
186     // RowsAffected implements Result for an INSERT or UPDATE op
187     // which mutates a number of rows.
188     type RowsAffected int64
189

```

```

190 var _ Result = RowsAffected(0)
191
192 func (RowsAffected) LastInsertId() (int64, error) {
193     return 0, errors.New("no LastInsertId available")
194 }
195
196 func (v RowsAffected) RowsAffected() (int64, error) {
197     return int64(v), nil
198 }
199
200 // ResultNoRows is a pre-defined Result for drivers to return
201 // command (such as a CREATE TABLE) succeeds. It returns an
202 // LastInsertId and RowsAffected.
203 var ResultNoRows noRows
204
205 type noRows struct{}
206
207 var _ Result = noRows{}
208
209 func (noRows) LastInsertId() (int64, error) {
210     return 0, errors.New("no LastInsertId available after")
211 }
212
213 func (noRows) RowsAffected() (int64, error) {
214     return 0, errors.New("no RowsAffected available after")
215 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/database/sql/driver/types.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package driver
6
7 import (
8     "fmt"
9     "reflect"
10    "strconv"
11    "time"
12 )
13
14 // ValueConverter is the interface providing the ConvertValue
15 //
16 // Various implementations of ValueConverter are provided by
17 // driver package to provide consistent implementations of c
18 // between drivers. The ValueConverters have several uses:
19 //
20 // * converting from the Value types as provided by the sql
21 //   into a database table's specific column type and makin
22 //   fits, such as making sure a particular int64 fits in a
23 //   table's uint16 column.
24 //
25 // * converting a value as given from the database into one
26 //   driver Value types.
27 //
28 // * by the sql package, for converting from a driver's Val
29 //   to a user's type in a scan.
30 type ValueConverter interface {
31     // ConvertValue converts a value to a driver Value.
32     ConvertValue(v interface{}) (Value, error)
33 }
34
35 // Valuer is the interface providing the Value method.
36 //
37 // Types implementing Valuer interface are able to convert
38 // themselves to a driver Value.
39 type Valuer interface {
40     // Value returns a driver Value.
41     Value() (Value, error)
```

```

42 }
43
44 // Bool is a ValueConverter that converts input values to bo
45 //
46 // The conversion rules are:
47 // - booleans are returned unchanged
48 // - for integer types,
49 //     1 is true
50 //     0 is false,
51 //     other integers are an error
52 // - for strings and []byte, same rules as strconv.ParseBoo
53 // - all other types are an error
54 var Bool boolType
55
56 type boolType struct{}
57
58 var _ ValueConverter = boolType{}
59
60 func (boolType) String() string { return "Bool" }
61
62 func (boolType) ConvertValue(src interface{}) (Value, error)
63     switch s := src.(type) {
64     case bool:
65         return s, nil
66     case string:
67         b, err := strconv.ParseBool(s)
68         if err != nil {
69             return nil, fmt.Errorf("sql/driver:
70         }
71         return b, nil
72     case []byte:
73         b, err := strconv.ParseBool(string(s))
74         if err != nil {
75             return nil, fmt.Errorf("sql/driver:
76         }
77         return b, nil
78     }
79
80     sv := reflect.ValueOf(src)
81     switch sv.Kind() {
82     case reflect.Int, reflect.Int8, reflect.Int16, refle
83         iv := sv.Int()
84         if iv == 1 || iv == 0 {
85             return iv == 1, nil
86         }
87         return nil, fmt.Errorf("sql/driver: couldn't
88     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
89         uv := sv.Uint()
90         if uv == 1 || uv == 0 {
91             return uv == 1, nil

```

```

92         }
93         return nil, fmt.Errorf("sql/driver: couldn't
94     }
95
96     return nil, fmt.Errorf("sql/driver: couldn't convert
97 }
98
99 // Int32 is a ValueConverter that converts input values to i
100 // respecting the limits of an int32 value.
101 var Int32 int32Type
102
103 type int32Type struct{}
104
105 var _ ValueConverter = int32Type{}
106
107 func (int32Type) ConvertValue(v interface{}) (Value, error)
108     rv := reflect.ValueOf(v)
109     switch rv.Kind() {
110     case reflect.Int, reflect.Int8, reflect.Int16, refle
111         i64 := rv.Int()
112         if i64 > (1<<31)-1 || i64 < -(1<<31) {
113             return nil, fmt.Errorf("sql/driver:
114         }
115         return i64, nil
116     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
117         u64 := rv.Uint()
118         if u64 > (1<<31)-1 {
119             return nil, fmt.Errorf("sql/driver:
120         }
121         return int64(u64), nil
122     case reflect.String:
123         i, err := strconv.Atoi(rv.String())
124         if err != nil {
125             return nil, fmt.Errorf("sql/driver:
126         }
127         return int64(i), nil
128     }
129     return nil, fmt.Errorf("sql/driver: unsupported valu
130 }
131
132 // String is a ValueConverter that converts its input to a s
133 // If the value is already a string or []byte, it's unchange
134 // If the value is of another type, conversion to string is
135 // with fmt.Sprintf("%v", v).
136 var String stringType
137
138 type stringType struct{}
139
140 func (stringType) ConvertValue(v interface{}) (Value, error)

```

```

141         switch v.(type) {
142             case string, []byte:
143                 return v, nil
144             }
145         return fmt.Sprintf("%v", v), nil
146     }
147
148     // Null is a type that implements ValueConverter by allowing
149     // values but otherwise delegating to another ValueConverter
150     type Null struct {
151         Converter ValueConverter
152     }
153
154     func (n Null) ConvertValue(v interface{}) (Value, error) {
155         if v == nil {
156             return nil, nil
157         }
158         return n.Converter.ConvertValue(v)
159     }
160
161     // NotNull is a type that implements ValueConverter by disal
162     // values but otherwise delegating to another ValueConverter
163     type NotNull struct {
164         Converter ValueConverter
165     }
166
167     func (n NotNull) ConvertValue(v interface{}) (Value, error) {
168         if v == nil {
169             return nil, fmt.Errorf("nil value not allowe
170         }
171         return n.Converter.ConvertValue(v)
172     }
173
174     // IsValue reports whether v is a valid Value parameter type
175     // Unlike IsScanValue, IsValue permits the string type.
176     func IsValue(v interface{}) bool {
177         if IsScanValue(v) {
178             return true
179         }
180         if _, ok := v.(string); ok {
181             return true
182         }
183         return false
184     }
185
186     // IsScanValue reports whether v is a valid Value scan type.
187     // Unlike IsValue, IsScanValue does not permit the string ty
188     func IsScanValue(v interface{}) bool {
189         if v == nil {

```

```

190         return true
191     }
192     switch v.(type) {
193     case int64, float64, []byte, bool, time.Time:
194         return true
195     }
196     return false
197 }
198
199 // DefaultParameterConverter is the default implementation of
200 // ValueConverter that's used when a Stmt doesn't implement
201 // ColumnConverter.
202 //
203 // DefaultParameterConverter returns the given value directly
204 // if IsValue(value). Otherwise integer types are converted to
205 // int64, floats to float64, and strings to []byte. Otherwise
206 // an error.
207 var DefaultParameterConverter defaultConverter
208
209 type defaultConverter struct{}
210
211 var _ ValueConverter = defaultConverter{}
212
213 func (defaultConverter) ConvertValue(v interface{}) (Value,
214     error) {
215     if IsValue(v) {
216         return v, nil
217     }
218     if svi, ok := v.(Valuer); ok {
219         sv, err := svi.Value()
220         if err != nil {
221             return nil, err
222         }
223         if !IsValue(sv) {
224             return nil, fmt.Errorf("non-Value type")
225         }
226         return sv, nil
227     }
228
229     rv := reflect.ValueOf(v)
230     switch rv.Kind() {
231     case reflect.Ptr:
232         // indirect pointers
233         if rv.IsNil() {
234             return nil, nil
235         } else {
236             return defaultConverter{}.ConvertValue(rv.Elem().Interface())
237         }
238     case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32,
239         reflect.Int64, reflect.Uint, reflect.Uint8, reflect.Uint16,

```

```
240     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
241         return int64(rv.Uint()), nil
242     case reflect.Uint64:
243         u64 := rv.Uint()
244         if u64 >= 1<<63 {
245             return nil, fmt.Errorf("uint64 value
246         }
247         return int64(u64), nil
248     case reflect.Float32, reflect.Float64:
249         return rv.Float(), nil
250     }
251     return nil, fmt.Errorf("unsupported type %T, a %s",
252 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/dwarf/buf.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Buffered reading and decoding of DWARF data streams.
6
7 package dwarf
8
9 import (
10     "encoding/binary"
11     "strconv"
12 )
13
14 // Data buffer being decoded.
15 type buf struct {
16     dwarf    *Data
17     order    binary.ByteOrder
18     name     string
19     off      Offset
20     data     []byte
21     addrsize int
22     err      error
23 }
24
25 func makeBuf(d *Data, name string, off Offset, data []byte,
26             return buf{d, d.order, name, off, data, addrsize, ni
27 }
28
29 func (b *buf) uint8() uint8 {
30     if len(b.data) < 1 {
31         b.error("underflow")
32         return 0
33     }
34     val := b.data[0]
35     b.data = b.data[1:]
36     b.off++
37     return val
38 }
39
40 func (b *buf) bytes(n int) []byte {
41     if len(b.data) < n {
```

```

42             b.error("underflow")
43             return nil
44         }
45         data := b.data[0:n]
46         b.data = b.data[n:]
47         b.off += Offset(n)
48         return data
49     }
50
51     func (b *buf) skip(n int) { b.bytes(n) }
52
53     func (b *buf) string() string {
54         for i := 0; i < len(b.data); i++ {
55             if b.data[i] == 0 {
56                 s := string(b.data[0:i])
57                 b.data = b.data[i+1:]
58                 b.off += Offset(i + 1)
59                 return s
60             }
61         }
62         b.error("underflow")
63         return ""
64     }
65
66     func (b *buf) uint16() uint16 {
67         a := b.bytes(2)
68         if a == nil {
69             return 0
70         }
71         return b.order.Uint16(a)
72     }
73
74     func (b *buf) uint32() uint32 {
75         a := b.bytes(4)
76         if a == nil {
77             return 0
78         }
79         return b.order.Uint32(a)
80     }
81
82     func (b *buf) uint64() uint64 {
83         a := b.bytes(8)
84         if a == nil {
85             return 0
86         }
87         return b.order.Uint64(a)
88     }
89
90     // Read a varint, which is 7 bits per byte, little endian.
91     // the 0x80 bit means read another byte.

```

```

92 func (b *buf) varint() (c uint64, bits uint) {
93     for i := 0; i < len(b.data); i++ {
94         byte := b.data[i]
95         c |= uint64(byte&0x7F) << bits
96         bits += 7
97         if byte&0x80 == 0 {
98             b.off += Offset(i + 1)
99             b.data = b.data[i+1:]
100            return c, bits
101        }
102    }
103    return 0, 0
104 }
105
106 // Unsigned int is just a varint.
107 func (b *buf) uint() uint64 {
108     x, _ := b.varint()
109     return x
110 }
111
112 // Signed int is a sign-extended varint.
113 func (b *buf) int() int64 {
114     ux, bits := b.varint()
115     x := int64(ux)
116     if x&(1<<(bits-1)) != 0 {
117         x |= -1 << bits
118     }
119     return x
120 }
121
122 // Address-sized uint.
123 func (b *buf) addr() uint64 {
124     switch b.addrsz {
125     case 1:
126         return uint64(b.uint8())
127     case 2:
128         return uint64(b.uint16())
129     case 4:
130         return uint64(b.uint32())
131     case 8:
132         return uint64(b.uint64())
133     }
134     b.error("unknown address size")
135     return 0
136 }
137
138 func (b *buf) error(s string) {
139     if b.err == nil {
140         b.data = nil

```

```
141             b.err = DecodeError{b.name, b.off, s}
142         }
143     }
144
145     type DecodeError struct {
146         Name    string
147         Offset  Offset
148         Err     string
149     }
150
151     func (e DecodeError) Error() string {
152         return "decoding dwarf section " + e.Name + " at off
153     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/dwarf/const.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Constants
6
7 package dwarf
8
9 import "strconv"
10
11 // An Attr identifies the attribute type in a DWARF Entry's
12 type Attr uint32
13
14 const (
15     AttrSibling          Attr = 0x01
16     AttrLocation         Attr = 0x02
17     AttrName             Attr = 0x03
18     AttrOrdering         Attr = 0x09
19     AttrByteSize         Attr = 0x0B
20     AttrBitOffset        Attr = 0x0C
21     AttrBitSize          Attr = 0x0D
22     AttrStmtList         Attr = 0x10
23     AttrLowpc            Attr = 0x11
24     AttrHighpc           Attr = 0x12
25     AttrLanguage         Attr = 0x13
26     AttrDiscr            Attr = 0x15
27     AttrDiscrValue       Attr = 0x16
28     AttrVisibility       Attr = 0x17
29     AttrImport           Attr = 0x18
30     AttrStringLength     Attr = 0x19
31     AttrCommonRef        Attr = 0x1A
32     AttrCompDir          Attr = 0x1B
33     AttrConstValue       Attr = 0x1C
34     AttrContainingType   Attr = 0x1D
35     AttrDefaultValue     Attr = 0x1E
36     AttrInline           Attr = 0x20
37     AttrIsOptional       Attr = 0x21
38     AttrLowerBound       Attr = 0x22
39     AttrProducer         Attr = 0x25
40     AttrPrototyped       Attr = 0x27
41     AttrReturnAddr       Attr = 0x2A
```

```
42     AttrStartScope      Attr = 0x2C
43     AttrStrideSize      Attr = 0x2E
44     AttrUpperBound      Attr = 0x2F
45     AttrAbstractOrigin  Attr = 0x31
46     AttrAccessibility   Attr = 0x32
47     AttrAddrClass       Attr = 0x33
48     AttrArtificial      Attr = 0x34
49     AttrBaseTypes       Attr = 0x35
50     AttrCalling         Attr = 0x36
51     AttrCount           Attr = 0x37
52     AttrDataMemberLoc   Attr = 0x38
53     AttrDeclColumn      Attr = 0x39
54     AttrDeclFile        Attr = 0x3A
55     AttrDeclLine        Attr = 0x3B
56     AttrDeclaration     Attr = 0x3C
57     AttrDiscrList       Attr = 0x3D
58     AttrEncoding        Attr = 0x3E
59     AttrExternal        Attr = 0x3F
60     AttrFrameBase      Attr = 0x40
61     AttrFriend          Attr = 0x41
62     AttrIdentifierCase  Attr = 0x42
63     AttrMacroInfo       Attr = 0x43
64     AttrNameListItem    Attr = 0x44
65     AttrPriority         Attr = 0x45
66     AttrSegment         Attr = 0x46
67     AttrSpecification   Attr = 0x47
68     AttrStaticLink      Attr = 0x48
69     AttrType            Attr = 0x49
70     AttrUseLocation     Attr = 0x4A
71     AttrVarParam        Attr = 0x4B
72     AttrVirtuality      Attr = 0x4C
73     AttrVtableElemLoc  Attr = 0x4D
74     AttrAllocated       Attr = 0x4E
75     AttrAssociated      Attr = 0x4F
76     AttrDataLocation    Attr = 0x50
77     AttrStride          Attr = 0x51
78     AttrEntryPC         Attr = 0x52
79     AttrUseUTF8         Attr = 0x53
80     AttrExtension       Attr = 0x54
81     AttrRanges          Attr = 0x55
82     AttrTrampoline     Attr = 0x56
83     AttrCallColumn     Attr = 0x57
84     AttrCallFile        Attr = 0x58
85     AttrCallLine        Attr = 0x59
86     AttrDescription     Attr = 0x5A
87 )
88
89 var attrNames = [...]string{
90     AttrSibling:      "Sibling",
91     AttrLocation:     "Location",
```

92	AttrName:	"Name",
93	AttrOrdering:	"Ordering",
94	AttrByteSize:	"ByteSize",
95	AttrBitOffset:	"BitOffset",
96	AttrBitSize:	"BitSize",
97	AttrStmtList:	"StmtList",
98	AttrLowpc:	"Lowpc",
99	AttrHighpc:	"Highpc",
100	AttrLanguage:	"Language",
101	AttrDiscr:	"Discr",
102	AttrDiscrValue:	"DiscrValue",
103	AttrVisibility:	"Visibility",
104	AttrImport:	"Import",
105	AttrStringLength:	"StringLength",
106	AttrCommonRef:	"CommonRef",
107	AttrCompDir:	"CompDir",
108	AttrConstValue:	"ConstValue",
109	AttrContainingType:	"ContainingType",
110	AttrDefaultValue:	"DefaultValue",
111	AttrInline:	"Inline",
112	AttrIsOptional:	"IsOptional",
113	AttrLowerBound:	"LowerBound",
114	AttrProducer:	"Producer",
115	AttrPrototyped:	"Prototyped",
116	AttrReturnAddr:	"ReturnAddr",
117	AttrStartScope:	"StartScope",
118	AttrStrideSize:	"StrideSize",
119	AttrUpperBound:	"UpperBound",
120	AttrAbstractOrigin:	"AbstractOrigin",
121	AttrAccessibility:	"Accessibility",
122	AttrAddrClass:	"AddrClass",
123	AttrArtificial:	"Artificial",
124	AttrBaseTypes:	"BaseTypes",
125	AttrCalling:	"Calling",
126	AttrCount:	"Count",
127	AttrDataMemberLoc:	"DataMemberLoc",
128	AttrDeclColumn:	"DeclColumn",
129	AttrDeclFile:	"DeclFile",
130	AttrDeclLine:	"DeclLine",
131	AttrDeclaration:	"Declaration",
132	AttrDiscrList:	"DiscrList",
133	AttrEncoding:	"Encoding",
134	AttrExternal:	"External",
135	AttrFrameBase:	"FrameBase",
136	AttrFriend:	"Friend",
137	AttrIdentifierCase:	"IdentifierCase",
138	AttrMacroInfo:	"MacroInfo",
139	AttrNamelistItem:	"NamelistItem",
140	AttrPriority:	"Priority",

```

141     AttrSegment:         "Segment",
142     AttrSpecification:   "Specification",
143     AttrStaticLink:      "StaticLink",
144     AttrType:            "Type",
145     AttrUseLocation:     "UseLocation",
146     AttrVarParam:        "VarParam",
147     AttrVirtuality:      "Virtuality",
148     AttrVtableElemLoc:  "VtableElemLoc",
149     AttrAllocated:       "Allocated",
150     AttrAssociated:      "Associated",
151     AttrDataLocation:   "DataLocation",
152     AttrStride:          "Stride",
153     AttrEntryPC:         "EntryPC",
154     AttrUseUTF8:         "UseUTF8",
155     AttrExtension:       "Extension",
156     AttrRanges:          "Ranges",
157     AttrTrampoline:     "Trampoline",
158     AttrCallColumn:     "CallColumn",
159     AttrCallFile:        "CallFile",
160     AttrCallLine:        "CallLine",
161     AttrDescription:     "Description",
162 }
163
164 func (a Attr) String() string {
165     if int(a) < len(attrNames) {
166         s := attrNames[a]
167         if s != "" {
168             return s
169         }
170     }
171     return strconv.Itoa(int(a))
172 }
173
174 func (a Attr) GoString() string {
175     if int(a) < len(attrNames) {
176         s := attrNames[a]
177         if s != "" {
178             return "dwarf.Attr" + s
179         }
180     }
181     return "dwarf.Attr(" + strconv.FormatInt(int64(a), 1
182 )
183
184 // A format is a DWARF data encoding format.
185 type format uint32
186
187 const (
188     // value formats
189     formAddr      format = 0x01

```

```

190         formDwarfBlock2 format = 0x03
191         formDwarfBlock4 format = 0x04
192         formData2      format = 0x05
193         formData4      format = 0x06
194         formData8      format = 0x07
195         formString     format = 0x08
196         formDwarfBlock format = 0x09
197         formDwarfBlock1 format = 0x0A
198         formData1      format = 0x0B
199         formFlag       format = 0x0C
200         formSdata     format = 0x0D
201         formStrp      format = 0x0E
202         formUdata     format = 0x0F
203         formRefAddr   format = 0x10
204         formRef1      format = 0x11
205         formRef2      format = 0x12
206         formRef4      format = 0x13
207         formRef8      format = 0x14
208         formRefUdata  format = 0x15
209         formIndirect  format = 0x16
210     )
211
212     // A Tag is the classification (the type) of an Entry.
213     type Tag uint32
214
215     const (
216         TagArrayType      Tag = 0x01
217         TagClassType     Tag = 0x02
218         TagEntryPoint    Tag = 0x03
219         TagEnumerationType Tag = 0x04
220         TagFormalParameter Tag = 0x05
221         TagImportedDeclaration Tag = 0x08
222         TagLabel        Tag = 0x0A
223         TagLexDwarfBlock Tag = 0x0B
224         TagMember       Tag = 0x0D
225         TagPointerType  Tag = 0x0F
226         TagReferenceType Tag = 0x10
227         TagCompileUnit  Tag = 0x11
228         TagStringType   Tag = 0x12
229         TagStructType   Tag = 0x13
230         TagSubroutineType Tag = 0x15
231         TagTypedef      Tag = 0x16
232         TagUnionType    Tag = 0x17
233         TagUnspecifiedParameters Tag = 0x18
234         TagVariant      Tag = 0x19
235         TagCommonDwarfBlock Tag = 0x1A
236         TagCommonInclusion Tag = 0x1B
237         TagInheritance  Tag = 0x1C
238         TagInlinedSubroutine Tag = 0x1D
239         TagModule       Tag = 0x1E

```

```

240     TagPtrToMemberType      Tag = 0x1F
241     TagSetType              Tag = 0x20
242     TagSubrangeType        Tag = 0x21
243     TagWithStmt            Tag = 0x22
244     TagAccessDeclaration    Tag = 0x23
245     TagBaseType            Tag = 0x24
246     TagCatchDwarfBlock     Tag = 0x25
247     TagConstType          Tag = 0x26
248     TagConstant            Tag = 0x27
249     TagEnumerator          Tag = 0x28
250     TagFileType            Tag = 0x29
251     TagFriend              Tag = 0x2A
252     TagNameList            Tag = 0x2B
253     TagNameListItem        Tag = 0x2C
254     TagPackedType          Tag = 0x2D
255     TagSubprogram          Tag = 0x2E
256     TagTemplateTypeParameter Tag = 0x2F
257     TagTemplateValueParameter Tag = 0x30
258     TagThrownType          Tag = 0x31
259     TagTryDwarfBlock       Tag = 0x32
260     TagVariantPart         Tag = 0x33
261     TagVariable            Tag = 0x34
262     TagVolatileType        Tag = 0x35
263     TagDwarfProcedure      Tag = 0x36
264     TagRestrictType        Tag = 0x37
265     TagInterfaceType       Tag = 0x38
266     TagNamespace           Tag = 0x39
267     TagImportedModule      Tag = 0x3A
268     TagUnspecifiedType     Tag = 0x3B
269     TagPartialUnit         Tag = 0x3C
270     TagImportedUnit        Tag = 0x3D
271     TagMutableType         Tag = 0x3E
272 )
273
274 var tagNames = [...]string{
275     TagArrayType:      "ArrayType",
276     TagClassType:     "ClassType",
277     TagEntryPoint:    "EntryPoint",
278     TagEnumerationType: "EnumerationType",
279     TagFormalParameter: "FormalParameter",
280     TagImportedDeclaration: "ImportedDeclaration",
281     TagLabel:         "Label",
282     TagLexDwarfBlock: "LexDwarfBlock",
283     TagMember:        "Member",
284     TagPointerType:   "PointerType",
285     TagReferenceType: "ReferenceType",
286     TagCompileUnit:   "CompileUnit",
287     TagStringType:    "StringType",
288     TagStructType:    "StructType",

```

```

289     TagSubroutineType:      "SubroutineType",
290     TagTypedef:             "Typedef",
291     TagUnionType:           "UnionType",
292     TagUnspecifiedParameters: "UnspecifiedParameters",
293     TagVariant:             "Variant",
294     TagCommonDwarfBlock:    "CommonDwarfBlock",
295     TagCommonInclusion:      "CommonInclusion",
296     TagInheritance:         "Inheritance",
297     TagInlinedSubroutine:   "InlinedSubroutine",
298     TagModule:              "Module",
299     TagPtrToMemberType:     "PtrToMemberType",
300     TagSetType:             "SetType",
301     TagSubrangeType:        "SubrangeType",
302     TagWithStmt:            "WithStmt",
303     TagAccessDeclaration:   "AccessDeclaration",
304     TagBaseType:            "BaseType",
305     TagCatchDwarfBlock:     "CatchDwarfBlock",
306     TagConstType:           "ConstType",
307     TagConstant:            "Constant",
308     TagEnumerator:          "Enumerator",
309     TagFileType:            "FileType",
310     TagFriend:              "Friend",
311     TagNameList:            "NameList",
312     TagNameListItem:        "NameListItem",
313     TagPackedType:          "PackedType",
314     TagSubprogram:          "Subprogram",
315     TagTemplateTypeParameter: "TemplateTypeParameter",
316     TagTemplateValueParameter: "TemplateValueParameter",
317     TagThrownType:          "ThrownType",
318     TagTryDwarfBlock:       "TryDwarfBlock",
319     TagVariantPart:         "VariantPart",
320     TagVariable:            "Variable",
321     TagVolatileType:        "VolatileType",
322     TagDwarfProcedure:      "DwarfProcedure",
323     TagRestrictType:        "RestrictType",
324     TagInterfaceType:       "InterfaceType",
325     TagNamespace:           "Namespace",
326     TagImportedModule:      "ImportedModule",
327     TagUnspecifiedType:     "UnspecifiedType",
328     TagPartialUnit:         "PartialUnit",
329     TagImportedUnit:        "ImportedUnit",
330     TagMutableType:         "MutableType",
331 }
332
333 func (t Tag) String() string {
334     if int(t) < len(tagNames) {
335         s := tagNames[t]
336         if s != "" {
337             return s

```

```

338         }
339     }
340     return strconv.Itoa(int(t))
341 }
342
343 func (t Tag) GoString() string {
344     if int(t) < len(tagNames) {
345         s := tagNames[t]
346         if s != "" {
347             return "dwarf.Tag" + s
348         }
349     }
350     return "dwarf.Tag(" + strconv.FormatInt(int64(t), 10
351 )
352
353 // Location expression operators.
354 // The debug info encodes value locations like 8(R3)
355 // as a sequence of these op codes.
356 // This package does not implement full expressions;
357 // the opPlusUconst operator is expected by the type parser.
358 const (
359     opAddr      = 0x03 /* 1 op, const addr */
360     opDeref     = 0x06
361     opConst1u   = 0x08 /* 1 op, 1 byte const */
362     opConst1s   = 0x09 /* " signed */
363     opConst2u   = 0x0A /* 1 op, 2 byte const */
364     opConst2s   = 0x0B /* " signed */
365     opConst4u   = 0x0C /* 1 op, 4 byte const */
366     opConst4s   = 0x0D /* " signed */
367     opConst8u   = 0x0E /* 1 op, 8 byte const */
368     opConst8s   = 0x0F /* " signed */
369     opConstu    = 0x10 /* 1 op, LEB128 const */
370     opConsts    = 0x11 /* " signed */
371     opDup       = 0x12
372     opDrop      = 0x13
373     opOver      = 0x14
374     opPick      = 0x15 /* 1 op, 1 byte stack index */
375     opSwap      = 0x16
376     opRot       = 0x17
377     opXderef    = 0x18
378     opAbs       = 0x19
379     opAnd       = 0x1A
380     opDiv       = 0x1B
381     opMinus     = 0x1C
382     opMod       = 0x1D
383     opMul       = 0x1E
384     opNeg       = 0x1F
385     opNot       = 0x20
386     opOr        = 0x21
387     opPlus      = 0x22

```

```

388         opPlusUconst = 0x23 /* 1 op, ULEB128 addend */
389         opShl         = 0x24
390         opShr         = 0x25
391         opShra        = 0x26
392         opXor         = 0x27
393         opSkip        = 0x2F /* 1 op, signed 2-byte constant
394         opBra         = 0x28 /* 1 op, signed 2-byte constant
395         opEq          = 0x29
396         opGe          = 0x2A
397         opGt          = 0x2B
398         opLe          = 0x2C
399         opLt          = 0x2D
400         opNe          = 0x2E
401         opLit0        = 0x30
402         /* OpLitN = OpLit0 + N for N = 0..31 */
403         opReg0        = 0x50
404         /* OpRegN = OpReg0 + N for N = 0..31 */
405         opBreg0       = 0x70 /* 1 op, signed LEB128 constant */
406         /* OpBregN = OpBreg0 + N for N = 0..31 */
407         opRegx        = 0x90 /* 1 op, ULEB128 register */
408         opFbreg       = 0x91 /* 1 op, SLEB128 offset */
409         opBregx       = 0x92 /* 2 op, ULEB128 reg; SLEB128 of
410         opPiece       = 0x93 /* 1 op, ULEB128 size of piece */
411         opDerefSize  = 0x94 /* 1-byte size of data retrieved
412         opXderefSize = 0x95 /* 1-byte size of data retrieved
413         opNop         = 0x96
414         /* next four new in Dwarf v3 */
415         opPushObjAddr = 0x97
416         opCall12      = 0x98 /* 2-byte offset of DIE */
417         opCall14      = 0x99 /* 4-byte offset of DIE */
418         opCallRef     = 0x9A /* 4- or 8- byte offset of DIE
419         /* 0xE0-0xFF reserved for user-specific */
420     )
421
422     // Basic type encodings -- the value for AttrEncoding in a T
423     const (
424         encAddress      = 0x01
425         encBoolean      = 0x02
426         encComplexFloat = 0x03
427         encFloat        = 0x04
428         encSigned       = 0x05
429         encSignedChar   = 0x06
430         encUnsigned     = 0x07
431         encUnsignedChar = 0x08
432         encImaginaryFloat = 0x09
433     )

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/dwarf/entry.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // DWARF debug information entry parser.
6 // An entry is a sequence of data items of a given format.
7 // The first word in the entry is an index into what DWARF
8 // calls the ``abbreviation table.'' An abbreviation is rea
9 // just a type descriptor: it's an array of attribute tag/va
10
11 package dwarf
12
13 import "errors"
14
15 // a single entry's description: a sequence of attributes
16 type abbrev struct {
17     tag      Tag
18     children bool
19     field    []afield
20 }
21
22 type afield struct {
23     attr Attr
24     fmt  format
25 }
26
27 // a map from entry format ids to their descriptions
28 type abbrevTable map[uint32]abbrev
29
30 // ParseAbbrev returns the abbreviation table that starts at
31 // in the .debug_abbrev section.
32 func (d *Data) parseAbbrev(off uint32) (abbrevTable, error)
33     if m, ok := d.abbrevCache[off]; ok {
34         return m, nil
35     }
36
37     data := d.abbrev
38     if off > uint32(len(data)) {
39         data = nil
40     } else {
41         data = data[off:]
```

```

42     }
43     b := makeBuf(d, "abbrev", 0, data, 0)
44
45     // Error handling is simplified by the buf getters
46     // returning an endless stream of 0s after an error.
47     m := make(abbrevTable)
48     for {
49         // Table ends with id == 0.
50         id := uint32(b.uint())
51         if id == 0 {
52             break
53         }
54
55         // Walk over attributes, counting.
56         n := 0
57         b1 := b // Read from copy of b.
58         b1.uint()
59         b1.uint8()
60         for {
61             tag := b1.uint()
62             fmt := b1.uint()
63             if tag == 0 && fmt == 0 {
64                 break
65             }
66             n++
67         }
68         if b1.err != nil {
69             return nil, b1.err
70         }
71
72         // Walk over attributes again, this time write
73         var a abbrev
74         a.tag = Tag(b.uint())
75         a.children = b.uint8() != 0
76         a.field = make([]afield, n)
77         for i := range a.field {
78             a.field[i].attr = Attr(b.uint())
79             a.field[i].fmt = format(b.uint())
80         }
81         b.uint()
82         b.uint()
83
84         m[id] = a
85     }
86     if b.err != nil {
87         return nil, b.err
88     }
89     d.abbrevCache[off] = m
90     return m, nil
91 }

```

```

92
93 // An entry is a sequence of attribute/value pairs.
94 type Entry struct {
95     Offset    Offset // offset of Entry in DWARF info
96     Tag       Tag     // tag (kind of Entry)
97     Children  bool    // whether Entry is followed by chil
98     Field     []Field
99 }
100
101 // A Field is a single attribute/value pair in an Entry.
102 type Field struct {
103     Attr Attr
104     Val  interface{}
105 }
106
107 // Val returns the value associated with attribute Attr in E
108 // or nil if there is no such attribute.
109 //
110 // A common idiom is to merge the check for nil return with
111 // the check that the value has the expected dynamic type, a
112 //     v, ok := e.Val(AttrSibling).(int64);
113 //
114 func (e *Entry) Val(a Attr) interface{} {
115     for _, f := range e.Field {
116         if f.Attr == a {
117             return f.Val
118         }
119     }
120     return nil
121 }
122
123 // An Offset represents the location of an Entry within the
124 // (See Reader.Seek.)
125 type Offset uint32
126
127 // Entry reads a single entry from buf, decoding
128 // according to the given abbreviation table.
129 func (b *buf) entry(atab abbrevTable, ubase Offset) *Entry {
130     off := b.off
131     id := uint32(b.uint())
132     if id == 0 {
133         return &Entry{}
134     }
135     a, ok := atab[id]
136     if !ok {
137         b.error("unknown abbreviation table index")
138         return nil
139     }
140     e := &Entry{

```

```

141         Offset:    off,
142         Tag:       a.tag,
143         Children:  a.children,
144         Field:     make([]Field, len(a.field)),
145     }
146     for i := range e.Field {
147         e.Field[i].Attr = a.field[i].attr
148         fmt := a.field[i].fmt
149         if fmt == formIndirect {
150             fmt = format(b.uint())
151         }
152         var val interface{}
153         switch fmt {
154         default:
155             b.error("unknown entry attr format")
156
157         // address
158         case formAddr:
159             val = b.addr()
160
161         // block
162         case formDwarfBlock1:
163             val = b.bytes(int(b.uint8()))
164         case formDwarfBlock2:
165             val = b.bytes(int(b.uint16()))
166         case formDwarfBlock4:
167             val = b.bytes(int(b.uint32()))
168         case formDwarfBlock:
169             val = b.bytes(int(b.uint()))
170
171         // constant
172         case formData1:
173             val = int64(b.uint8())
174         case formData2:
175             val = int64(b.uint16())
176         case formData4:
177             val = int64(b.uint32())
178         case formData8:
179             val = int64(b.uint64())
180         case formSdata:
181             val = int64(b.int())
182         case formUdata:
183             val = int64(b.uint())
184
185         // flag
186         case formFlag:
187             val = b.uint8() == 1
188
189         // reference to other entry

```

```

190         case formRefAddr:
191             val = Offset(b.addr())
192         case formRef1:
193             val = Offset(b.uint8()) + ubase
194         case formRef2:
195             val = Offset(b.uint16()) + ubase
196         case formRef4:
197             val = Offset(b.uint32()) + ubase
198         case formRef8:
199             val = Offset(b.uint64()) + ubase
200         case formRefUdata:
201             val = Offset(b.uint()) + ubase
202
203         // string
204         case formString:
205             val = b.string()
206         case formStrp:
207             off := b.uint32() // offset into .de
208             if b.err != nil {
209                 return nil
210             }
211             b1 := makeBuf(b.dwarf, "str", 0, b.d
212             b1.skip(int(off))
213             val = b1.string()
214             if b1.err != nil {
215                 b.err = b1.err
216                 return nil
217             }
218         }
219         e.Field[i].Val = val
220     }
221     if b.err != nil {
222         return nil
223     }
224     return e
225 }
226
227 // A Reader allows reading Entry structures from a DWARF ``i
228 // The Entry structures are arranged in a tree. The Reader'
229 // return successive entries from a pre-order traversal of t
230 // If an entry has children, its Children field will be true
231 // follow, terminated by an Entry with Tag 0.
232 type Reader struct {
233     b          buf
234     d          *Data
235     err        error
236     unit       int
237     lastChildren bool // .Children of last entry retur
238     lastSibling Offset // .Val(AttrSibling) of last ent
239 }

```

```

240
241 // Reader returns a new Reader for Data.
242 // The reader is positioned at byte offset 0 in the DWARF ``
243 func (d *Data) Reader() *Reader {
244     r := &Reader{d: d}
245     r.Seek(0)
246     return r
247 }
248
249 // Seek positions the Reader at offset off in the encoded en
250 // Offset 0 can be used to denote the first entry.
251 func (r *Reader) Seek(off Offset) {
252     d := r.d
253     r.err = nil
254     r.lastChildren = false
255     if off == 0 {
256         if len(d.unit) == 0 {
257             return
258         }
259         u := &d.unit[0]
260         r.unit = 0
261         r.b = makeBuf(r.d, "info", u.off, u.data, u.
262         return
263     }
264
265     // TODO(rsc): binary search (maybe a new package)
266     var i int
267     var u *unit
268     for i = range d.unit {
269         u = &d.unit[i]
270         if u.off <= off && off < u.off+Offset(len(u.
271             r.unit = i
272             r.b = makeBuf(r.d, "info", off, u.da
273             return
274         }
275     }
276     r.err = errors.New("offset out of range")
277 }
278
279 // maybeNextUnit advances to the next unit if this one is fi
280 func (r *Reader) maybeNextUnit() {
281     for len(r.b.data) == 0 && r.unit+1 < len(r.d.unit) {
282         r.unit++
283         u := &r.d.unit[r.unit]
284         r.b = makeBuf(r.d, "info", u.off, u.data, u.
285     }
286 }
287
288 // Next reads the next entry from the encoded entry stream.

```

```

289 // It returns nil, nil when it reaches the end of the sectio
290 // It returns an error if the current offset is invalid or t
291 // offset cannot be decoded as a valid Entry.
292 func (r *Reader) Next() (*Entry, error) {
293     if r.err != nil {
294         return nil, r.err
295     }
296     r.maybeNextUnit()
297     if len(r.b.data) == 0 {
298         return nil, nil
299     }
300     u := &r.d.unit[r.unit]
301     e := r.b.entry(u.atable, u.base)
302     if r.b.err != nil {
303         r.err = r.b.err
304         return nil, r.err
305     }
306     if e != nil {
307         r.lastChildren = e.Children
308         if r.lastChildren {
309             r.lastSibling, _ = e.Val(AttrSibling)
310         }
311     } else {
312         r.lastChildren = false
313     }
314     return e, nil
315 }
316
317 // SkipChildren skips over the child entries associated with
318 // the last Entry returned by Next. If that Entry did not h
319 // children or Next has not been called, SkipChildren is a n
320 func (r *Reader) SkipChildren() {
321     if r.err != nil || !r.lastChildren {
322         return
323     }
324
325     // If the last entry had a sibling attribute,
326     // that attribute gives the offset of the next
327     // sibling, so we can avoid decoding the
328     // child subtrees.
329     if r.lastSibling >= r.b.off {
330         r.Seek(r.lastSibling)
331         return
332     }
333
334     for {
335         e, err := r.Next()
336         if err != nil || e == nil || e.Tag == 0 {
337             break

```

```
338         }
339         if e.Children {
340             r.SkipChildren()
341         }
342     }
343 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/dwarf/open.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package dwarf provides access to DWARF debugging informat
6 // executable files, as defined in the DWARF 2.0 Standard at
7 // http://dwarfstd.org/doc/dwarf-2.0.0.pdf
8 package dwarf
9
10 import "encoding/binary"
11
12 // Data represents the DWARF debugging information
13 // loaded from an executable file (for example, an ELF or Ma
14 type Data struct {
15     // raw data
16     abbrev    []byte
17     aranges   []byte
18     frame     []byte
19     info      []byte
20     line      []byte
21     pubnames  []byte
22     ranges    []byte
23     str       []byte
24
25     // parsed data
26     abbrevCache map[uint32]abbrevTable
27     addrsz      int
28     order       binary.ByteOrder
29     typeCache   map[Offset]Type
30     unit        []unit
31 }
32
33 // New returns a new Data object initialized from the given
34 // Rather than calling this function directly, clients shoul
35 // the DWARF method of the File type of the appropriate pack
36 // debug/macho, or debug/pe.
37 //
38 // The []byte arguments are the data from the corresponding
39 // in the object file; for example, for an ELF object, abbre
40 // the ".debug_abbrev" section.
41 func New(abbrev, aranges, frame, info, line, pubnames, range
```

```

42     d := &Data{
43         abbrev:      abbrev,
44         aranges:     aranges,
45         frame:       frame,
46         info:        info,
47         line:        line,
48         pubnames:    pubnames,
49         ranges:      ranges,
50         str:         str,
51         abbrevCache: make(map[uint32]abbrevTable),
52         typeCache:   make(map[Offset]Type),
53     }
54
55     // Sniff .debug_info to figure out byte order.
56     // bytes 4:6 are the version, a tiny 16-bit number (
57     if len(d.info) < 6 {
58         return nil, DecodeError{"info", Offset(len(d
59     }
60     x, y := d.info[4], d.info[5]
61     switch {
62     case x == 0 && y == 0:
63         return nil, DecodeError{"info", 4, "unsupport
64     case x == 0:
65         d.order = binary.BigEndian
66     case y == 0:
67         d.order = binary.LittleEndian
68     default:
69         return nil, DecodeError{"info", 4, "cannot d
70     }
71
72     u, err := d.parseUnits()
73     if err != nil {
74         return nil, err
75     }
76     d.unit = u
77     return d, nil
78 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/dwarf/type.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // DWARF type information structures.
6 // The format is heavily biased toward C, but for simplicity
7 // the String methods use a pseudo-Go syntax.
8
9 package dwarf
10
11 import "strconv"
12
13 // A Type conventionally represents a pointer to any of the
14 // specific Type structures (CharType, StructType, etc.).
15 type Type interface {
16     Common() *CommonType
17     String() string
18     Size() int64
19 }
20
21 // A CommonType holds fields common to multiple types.
22 // If a field is not known or not applicable for a given typ
23 // the zero value is used.
24 type CommonType struct {
25     ByteSize int64 // size of value of this type, in by
26     Name      string // name that can be used to refer to
27 }
28
29 func (c *CommonType) Common() *CommonType { return c }
30
31 func (c *CommonType) Size() int64 { return c.ByteSize }
32
33 // Basic types
34
35 // A BasicType holds fields common to all basic types.
36 type BasicType struct {
37     CommonType
38     BitSize    int64
39     BitOffset  int64
40 }
41
```

```

42 func (b *BasicType) Basic() *BasicType { return b }
43
44 func (t *BasicType) String() string {
45     if t.Name != "" {
46         return t.Name
47     }
48     return "?"
49 }
50
51 // A CharType represents a signed character type.
52 type CharType struct {
53     BasicType
54 }
55
56 // A UcharType represents an unsigned character type.
57 type UcharType struct {
58     BasicType
59 }
60
61 // An IntType represents a signed integer type.
62 type IntType struct {
63     BasicType
64 }
65
66 // A UintType represents an unsigned integer type.
67 type UintType struct {
68     BasicType
69 }
70
71 // A FloatType represents a floating point type.
72 type FloatType struct {
73     BasicType
74 }
75
76 // A ComplexType represents a complex floating point type.
77 type ComplexType struct {
78     BasicType
79 }
80
81 // A BoolType represents a boolean type.
82 type BoolType struct {
83     BasicType
84 }
85
86 // An AddrType represents a machine address type.
87 type AddrType struct {
88     BasicType
89 }
90
91 // qualifiers

```

```

92
93 // A QualType represents a type that has the C/C++ "const",
94 type QualType struct {
95     CommonType
96     Qual string
97     Type Type
98 }
99
100 func (t *QualType) String() string { return t.Qual + " " + t
101
102 func (t *QualType) Size() int64 { return t.Type.Size() }
103
104 // An ArrayType represents a fixed size array type.
105 type ArrayType struct {
106     CommonType
107     Type          Type
108     StrideBitSize int64 // if > 0, number of bits to hol
109     Count         int64 // if == -1, an incomplete array
110 }
111
112 func (t *ArrayType) String() string {
113     return "[" + strconv.FormatInt(t.Count, 10) + "]" +
114 }
115
116 func (t *ArrayType) Size() int64 { return t.Count * t.Type.S
117
118 // A VoidType represents the C void type.
119 type VoidType struct {
120     CommonType
121 }
122
123 func (t *VoidType) String() string { return "void" }
124
125 // A PtrType represents a pointer type.
126 type PtrType struct {
127     CommonType
128     Type Type
129 }
130
131 func (t *PtrType) String() string { return "*" + t.Type.Stri
132
133 // A StructType represents a struct, union, or C++ class typ
134 type StructType struct {
135     CommonType
136     StructName string
137     Kind       string // "struct", "union", or "class".
138     Field      []*StructField
139     Incomplete bool // if true, struct, union, class is
140 }

```

```

141
142 // A StructField represents a field in a struct, union, or C
143 type StructField struct {
144     Name      string
145     Type      Type
146     ByteOffset int64
147     ByteSize  int64
148     BitOffset int64 // within the ByteSize bytes at Byt
149     BitSize   int64 // zero if not a bit field
150 }
151
152 func (t *StructType) String() string {
153     if t.StructName != "" {
154         return t.Kind + " " + t.StructName
155     }
156     return t.Defn()
157 }
158
159 func (t *StructType) Defn() string {
160     s := t.Kind
161     if t.StructName != "" {
162         s += " " + t.StructName
163     }
164     if t.Incomplete {
165         s += " /*incomplete*/"
166         return s
167     }
168     s += " {"
169     for i, f := range t.Field {
170         if i > 0 {
171             s += "; "
172         }
173         s += f.Name + " " + f.Type.String()
174         s += "@" + strconv.FormatInt(f.ByteOffset, 1
175         if f.BitSize > 0 {
176             s += " : " + strconv.FormatInt(f.Bit
177             s += "@" + strconv.FormatInt(f.BitOf
178         }
179     }
180     s += "}"
181     return s
182 }
183
184 // An EnumType represents an enumerated type.
185 // The only indication of its native integer type is its Byt
186 // (inside CommonType).
187 type EnumType struct {
188     CommonType
189     EnumName string

```

```

190         Val        []*EnumValue
191     }
192
193     // An EnumValue represents a single enumeration value.
194     type EnumValue struct {
195         Name string
196         Val  int64
197     }
198
199     func (t *EnumType) String() string {
200         s := "enum"
201         if t.EnumName != "" {
202             s += " " + t.EnumName
203         }
204         s += " {"
205         for i, v := range t.Val {
206             if i > 0 {
207                 s += "; "
208             }
209             s += v.Name + "=" + strconv.FormatInt(v.Val,
210             }
211             s += "}"
212             return s
213         }
214
215     // A FuncType represents a function type.
216     type FuncType struct {
217         CommonType
218         ReturnType Type
219         ParamType  []Type
220     }
221
222     func (t *FuncType) String() string {
223         s := "func("
224         for i, t := range t.ParamType {
225             if i > 0 {
226                 s += ", "
227             }
228             s += t.String()
229         }
230         s += ")"
231         if t.ReturnType != nil {
232             s += " " + t.ReturnType.String()
233         }
234         return s
235     }
236
237     // A DotDotDotType represents the variadic ... function para
238     type DotDotDotType struct {
239         CommonType

```

```

240 }
241
242 func (t *DotDotDotType) String() string { return "..."}
243
244 // A TypedefType represents a named type.
245 type TypedefType struct {
246     CommonType
247     Type Type
248 }
249
250 func (t *TypedefType) String() string { return t.Name }
251
252 func (t *TypedefType) Size() int64 { return t.Type.Size() }
253
254 func (d *Data) Type(off Offset) (Type, error) {
255     if t, ok := d.typeCache[off]; ok {
256         return t, nil
257     }
258
259     r := d.Reader()
260     r.Seek(off)
261     e, err := r.Next()
262     if err != nil {
263         return nil, err
264     }
265     if e == nil || e.Offset != off {
266         return nil, DecodeError{"info", off, "no typ
267     }
268
269     // Parse type from Entry.
270     // Must always set d.typeCache[off] before calling
271     // d.Type recursively, to handle circular types corr
272     var typ Type
273
274     // Get next child; set err if error happens.
275     next := func() *Entry {
276         if !e.Children {
277             return nil
278         }
279         kid, err1 := r.Next()
280         if err1 != nil {
281             err = err1
282             return nil
283         }
284         if kid == nil {
285             err = DecodeError{"info", r.b.off, "
286         }
287         return nil
288     }
289     if kid.Tag == 0 {

```

```

289         return nil
290     }
291     return kid
292 }
293
294 // Get Type referred to by Entry's AttrType field.
295 // Set err if error happens. Not having a type is a
296 typeOf := func(e *Entry) Type {
297     toff, ok := e.Val(AttrType).(Offset)
298     if !ok {
299         // It appears that no Type means "vo
300         return new(VoidType)
301     }
302     var t Type
303     if t, err = d.Type(toff); err != nil {
304         return nil
305     }
306     return t
307 }
308
309 switch e.Tag {
310 case TagArrayType:
311     // Multi-dimensional array. (DWARF v2 §5.4)
312     // Attributes:
313     //     AttrType:subtype [required]
314     //     AttrStrideSize: size in bits of each
315     //     AttrByteSize: size of entire array
316     // Children:
317     //     TagSubrangeType or TagEnumerationTyp
318     //     dimensions are in left to right orde
319     t := new(ArrayType)
320     typ = t
321     d.typeCache[off] = t
322     if t.Type = typeOf(e); err != nil {
323         goto Error
324     }
325     t.StrideBitSize, _ = e.Val(AttrStrideSize).(
326
327     // Accumulate dimensions,
328     ndim := 0
329     for kid := next(); kid != nil; kid = next()
330         // TODO(rsc): Can also be TagEnumera
331         // but haven't seen that in the wild
332         switch kid.Tag {
333         case TagSubrangeType:
334             max, ok := kid.Val(AttrUpper
335             if !ok {
336                 max = -2 // Count ==
337             }

```

```

338         if ndim == 0 {
339             t.Count = max + 1
340         } else {
341             // Multidimensional
342             // Create new array
343             t.Type = &ArrayType{
344                 }
345             ndim++
346         case TagEnumerationType:
347             err = DecodeError{"info", ki
348             goto Error
349         }
350     }
351     if ndim == 0 {
352         // LLVM generates this for x[].
353         t.Count = -1
354     }
355
356     case TagBaseType:
357         // Basic type. (DWARF v2 §5.1)
358         // Attributes:
359         //     AttrName: name of base type in progr
360         //     AttrEncoding: encoding value for typ
361         //     AttrByteSize: size of type in bytes
362         //     AttrBitOffset: for sub-byte types, s
363         //     AttrBitSize: for sub-byte types, bit
364         name, _ := e.Val(AttrName).(string)
365         enc, ok := e.Val(AttrEncoding).(int64)
366         if !ok {
367             err = DecodeError{"info", e.Offset,
368             goto Error
369         }
370         switch enc {
371         default:
372             err = DecodeError{"info", e.Offset,
373             goto Error
374
375         case encAddress:
376             typ = new(AddrType)
377         case encBoolean:
378             typ = new(BoolType)
379         case encComplexFloat:
380             typ = new(ComplexType)
381         case encFloat:
382             typ = new(FloatType)
383         case encSigned:
384             typ = new(IntType)
385         case encUnsigned:
386             typ = new(UintType)
387         case encSignedChar:

```

```

388         typ = new(CharType)
389     case encUnsignedChar:
390         typ = new(UcharType)
391     }
392     d.typeCache[off] = typ
393     t := typ.(interface {
394         Basic() *BasicType
395     }).Basic()
396     t.Name = name
397     t.BitSize, _ = e.Val(AttrBitSize).(int64)
398     t.BitOffset, _ = e.Val(AttrBitOffset).(int64)
399
400     case TagClassType, TagStructType, TagUnionType:
401         // Structure, union, or class type. (DWARF
402         // Attributes:
403         //     AttrName: name of struct, union, or
404         //     AttrByteSize: byte size [required]
405         //     AttrDeclaration: if true, struct/uni
406         // Children:
407         //     TagMember to describe one member.
408         //     AttrName: name of member [re
409         //     AttrType: type of member [re
410         //     AttrByteSize: size in bytes
411         //     AttrBitOffset: bit offset wi
412         //     AttrBitSize: bit size for bi
413         //     AttrDataMemberLoc: location
414         // There is much more to handle C++, all ign
415         t := new(StructType)
416         typ = t
417         d.typeCache[off] = t
418         switch e.Tag {
419         case TagClassType:
420             t.Kind = "class"
421         case TagStructType:
422             t.Kind = "struct"
423         case TagUnionType:
424             t.Kind = "union"
425         }
426         t.StructName, _ = e.Val(AttrName).(string)
427         t.Incomplete = e.Val(AttrDeclaration) != nil
428         t.Field = make([]*StructField, 0, 8)
429         var lastFieldType Type
430         var lastFieldBitOffset int64
431         for kid := next(); kid != nil; kid = next()
432             if kid.Tag == TagMember {
433                 f := new(StructField)
434                 if f.Type = typeOf(kid); err
435                     goto Error
436             }

```

```

437         if loc, ok := kid.Val(AttrDa
438             b := makeBuf(d, "loc
439             if b.uint8() != opPl
440                 err = Decode
441                 goto Error
442             }
443             f.ByteOffset = int64
444             if b.err != nil {
445                 err = b.err
446                 goto Error
447             }
448         }
449
450         haveBitOffset := false
451         f.Name, _ = kid.Val(AttrName
452         f.ByteSize, _ = kid.Val(Attr
453         f.BitOffset, haveBitOffset =
454         f.BitSize, _ = kid.Val(AttrB
455         t.Field = append(t.Field, f)
456
457         bito := f.BitOffset
458         if !haveBitOffset {
459             bito = f.ByteOffset
460         }
461         if bito == lastFieldBitOffse
462             // Last field was ze
463             // (DWARF writes out
464             zeroArray(lastFieldT
465         }
466         lastFieldType = f.Type
467         lastFieldBitOffset = bito
468     }
469 }
470 if t.Kind != "union" {
471     b, ok := e.Val(AttrByteSize).(int64)
472     if ok && b*8 == lastFieldBitOffset {
473         // Final field must be zero
474         zeroArray(lastFieldType)
475     }
476 }
477
478 case TagConstType, TagVolatileType, TagRestrictType:
479     // Type modifier (DWARF v2 §5.2)
480     // Attributes:
481     //     AttrType: subtype
482     t := new(QualType)
483     typ = t
484     d.typeCache[off] = t
485     if t.Type = typeOf(e); err != nil {

```

```

486         goto Error
487     }
488     switch e.Tag {
489     case TagConstType:
490         t.Qual = "const"
491     case TagRestrictType:
492         t.Qual = "restrict"
493     case TagVolatileType:
494         t.Qual = "volatile"
495     }
496
497     case TagEnumerationType:
498         // Enumeration type (DWARF v2 §5.6)
499         // Attributes:
500         //     AttrName: enum name if any
501         //     AttrByteSize: bytes required to repr
502         // Children:
503         //     TagEnumerator:
504         //         AttrName: name of constant
505         //         AttrConstValue: value of con
506         t := new(EnumType)
507         typ = t
508         d.typeCache[off] = t
509         t.EnumName, _ = e.Val(AttrName).(string)
510         t.Val = make([]*EnumValue, 0, 8)
511         for kid := next(); kid != nil; kid = next()
512             if kid.Tag == TagEnumerator {
513                 f := new(EnumValue)
514                 f.Name, _ = kid.Val(AttrName)
515                 f.Val, _ = kid.Val(AttrConst
516                 n := len(t.Val)
517                 if n >= cap(t.Val) {
518                     val := make([]*EnumV
519                     copy(val, t.Val)
520                     t.Val = val
521                 }
522                 t.Val = t.Val[0 : n+1]
523                 t.Val[n] = f
524             }
525         }
526
527     case TagPointerType:
528         // Type modifier (DWARF v2 §5.2)
529         // Attributes:
530         //     AttrType: subtype [not required! vo
531         //     AttrAddrClass: address class [ignore
532         t := new(PtrType)
533         typ = t
534         d.typeCache[off] = t
535         if e.Val(AttrType) == nil {

```

```

536             t.Type = &VoidType{}
537             break
538         }
539         t.Type = typeOf(e)
540
541     case TagSubroutineType:
542         // Subroutine type. (DWARF v2 §5.7)
543         // Attributes:
544         //     AttrType: type of return value if an
545         //     AttrName: possible name of type [ign
546         //     AttrPrototyped: whether used ANSI C
547         // Children:
548         //     TagFormalParameter: typed parameter
549         //     AttrType: type of parameter
550         //     TagUnspecifiedParameter: final ...
551         t := new(FuncType)
552         typ = t
553         d.typeCache[off] = t
554         if t.ReturnType = typeOf(e); err != nil {
555             goto Error
556         }
557         t.ParamType = make([]Type, 0, 8)
558         for kid := next(); kid != nil; kid = next()
559             var tkid Type
560             switch kid.Tag {
561             default:
562                 continue
563             case TagFormalParameter:
564                 if tkid = typeOf(kid); err != nil {
565                     goto Error
566                 }
567             case TagUnspecifiedParameters:
568                 tkid = &DotDotDotType{}
569             }
570             t.ParamType = append(t.ParamType, tkid)
571         }
572
573     case TagTypedef:
574         // Typedef (DWARF v2 §5.3)
575         // Attributes:
576         //     AttrName: name [required]
577         //     AttrType: type definition [required]
578         t := new(TypedefType)
579         typ = t
580         d.typeCache[off] = t
581         t.Name, _ = e.Val(AttrName).(string)
582         t.Type = typeOf(e)
583     }
584

```

```

585         if err != nil {
586             goto Error
587         }
588     }
589     {
590         b, ok := e.Val(AttrByteSize).(int64)
591         if !ok {
592             b = -1
593         }
594         typ.Common().ByteSize = b
595     }
596     return typ, nil
597
598 Error:
599     // If the parse fails, take the type out of the cach
600     // so that the next call with this offset doesn't hi
601     // the cache and return success.
602     delete(d.typeCache, off)
603     return nil, err
604 }
605
606 func zeroArray(t Type) {
607     for {
608         at, ok := t.(*ArrayType)
609         if !ok {
610             break
611         }
612         at.Count = 0
613         t = at.Type
614     }
615 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/dwarf/unit.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package dwarf
6
7 import "strconv"
8
9 // DWARF debug info is split into a sequence of compilation
10 // Each unit has its own abbreviation table and address size
11
12 type unit struct {
13     base    Offset // byte offset of header within the
14     off     Offset // byte offset of data within the ag
15     data    []byte
16     atable  abbrevTable
17     addrsize int
18 }
19
20 func (d *Data) parseUnits() ([]unit, error) {
21     // Count units.
22     nunit := 0
23     b := makeBuf(d, "info", 0, d.info, 0)
24     for len(b.data) > 0 {
25         b.skip(int(b.uint32()))
26         nunit++
27     }
28     if b.err != nil {
29         return nil, b.err
30     }
31
32     // Again, this time writing them down.
33     b = makeBuf(d, "info", 0, d.info, 0)
34     units := make([]unit, nunit)
35     for i := range units {
36         u := &units[i]
37         u.base = b.off
38         n := b.uint32()
39         if vers := b.uint16(); vers != 2 {
40             b.error("unsupported DWARF version ")
41             break

```

```
42         }
43         atable, err := d.parseAbbrev(b.uint32())
44         if err != nil {
45             if b.err == nil {
46                 b.err = err
47             }
48             break
49         }
50         u.atable = atable
51         u.addrsize = int(b.uint8())
52         u.off = b.off
53         u.data = b.bytes(int(n - (2 + 4 + 1)))
54     }
55     if b.err != nil {
56         return nil, b.err
57     }
58     return units, nil
59 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/debug/elf/elf.go

```
1  /*
2  * ELF constants and data structures
3  *
4  * Derived from:
5  * $FreeBSD: src/sys/sys/elf32.h,v 1.8.14.1 2005/12/30 22:13
6  * $FreeBSD: src/sys/sys/elf64.h,v 1.10.14.1 2005/12/30 22:1
7  * $FreeBSD: src/sys/sys/elf_common.h,v 1.15.8.1 2005/12/30
8  * $FreeBSD: src/sys/alpha/include/elf.h,v 1.14 2003/09/25 0
9  * $FreeBSD: src/sys/amd64/include/elf.h,v 1.18 2004/08/03 0
10 * $FreeBSD: src/sys/arm/include/elf.h,v 1.5.2.1 2006/06/30
11 * $FreeBSD: src/sys/i386/include/elf.h,v 1.16 2004/08/02 19
12 * $FreeBSD: src/sys/powerpc/include/elf.h,v 1.7 2004/11/02
13 * $FreeBSD: src/sys/sparc64/include/elf.h,v 1.12 2003/09/25
14 *
15 * Copyright (c) 1996-1998 John D. Polstra. All rights rese
16 * Copyright (c) 2001 David E. O'Brien
17 * Portions Copyright 2009 The Go Authors. All rights reser
18 *
19 * Redistribution and use in source and binary forms, with o
20 * modification, are permitted provided that the following c
21 * are met:
22 * 1. Redistributions of source code must retain the above c
23 * notice, this list of conditions and the following disc
24 * 2. Redistributions in binary form must reproduce the abov
25 * notice, this list of conditions and the following disc
26 * documentation and/or other materials provided with the
27 *
28 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS
29 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIM
30 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A P
31 * ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBU
32 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
33 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SU
34 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS I
35 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
36 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) AR
37 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE P
38 * SUCH DAMAGE.
39 */
40
41 package elf
42
43 import "strconv"
44
```

```

45  /*
46   * Constants
47   */
48
49 // Indexes into the Header.Ident array.
50 const (
51     EI_CLASS      = 4 /* Class of machine. */
52     EI_DATA       = 5 /* Data format. */
53     EI_VERSION    = 6 /* ELF format version. */
54     EI_OSABI      = 7 /* Operating system / ABI identif
55     EI_ABIVERSION = 8 /* ABI version */
56     EI_PAD        = 9 /* Start of padding (per SVR4 ABI
57     EI_NIDENT     = 16 /* Size of e_ident array. */
58 )
59
60 // Initial magic number for ELF files.
61 const ELFMAG = "\177ELF"
62
63 // Version is found in Header.Ident[EI_VERSION] and Header.v
64 type Version byte
65
66 const (
67     EV_NONE      Version = 0
68     EV_CURRENT   Version = 1
69 )
70
71 var versionStrings = []intName{
72     {0, "EV_NONE"},
73     {1, "EV_CURRENT"},
74 }
75
76 func (i Version) String() string { return stringName(uint3
77 func (i Version) GoString() string { return stringName(uint3
78
79 // Class is found in Header.Ident[EI_CLASS] and Header.Class
80 type Class byte
81
82 const (
83     ELFCLASSNONE Class = 0 /* Unknown class. */
84     ELFCLASS32   Class = 1 /* 32-bit architecture. */
85     ELFCLASS64   Class = 2 /* 64-bit architecture. */
86 )
87
88 var classStrings = []intName{
89     {0, "ELFCLASSNONE"},
90     {1, "ELFCLASS32"},
91     {2, "ELFCLASS64"},
92 }
93
94 func (i Class) String() string { return stringName(uint32(

```

```

95 func (i Class) GoString() string { return stringName(uint32(
96
97 // Data is found in Header.Ident[EI_DATA] and Header.Data.
98 type Data byte
99
100 const (
101     ELFDATANONE Data = 0 /* Unknown data format. */
102     ELFDATA2LSB Data = 1 /* 2's complement little-endian
103     ELFDATA2MSB Data = 2 /* 2's complement big-endian. *
104 )
105
106 var dataStrings = []intName{
107     {0, "ELFDATANONE"},
108     {1, "ELFDATA2LSB"},
109     {2, "ELFDATA2MSB"},
110 }
111
112 func (i Data) String() string { return stringName(uint32(i
113 func (i Data) GoString() string { return stringName(uint32(i
114
115 // OSABI is found in Header.Ident[EI_OSABI] and Header.OSABI
116 type OSABI byte
117
118 const (
119     ELFOSABI_NONE      OSABI = 0 /* UNIX System V ABI
120     ELFOSABI_HPUX      OSABI = 1 /* HP-UX operating s
121     ELFOSABI_NETBSD    OSABI = 2 /* NetBSD */
122     ELFOSABI_LINUX     OSABI = 3 /* GNU/Linux */
123     ELFOSABI_HURD      OSABI = 4 /* GNU/Hurd */
124     ELFOSABI_86OPEN    OSABI = 5 /* 86Open common IA3
125     ELFOSABI_SOLARIS   OSABI = 6 /* Solaris */
126     ELFOSABI_AIX       OSABI = 7 /* AIX */
127     ELFOSABI_IRIX      OSABI = 8 /* IRIX */
128     ELFOSABI_FREEBSD   OSABI = 9 /* FreeBSD */
129     ELFOSABI_TRU64     OSABI = 10 /* TRU64 UNIX */
130     ELFOSABI_MODESTO   OSABI = 11 /* Novell Modesto */
131     ELFOSABI_OPENBSD   OSABI = 12 /* OpenBSD */
132     ELFOSABI_OPENVMS   OSABI = 13 /* Open VMS */
133     ELFOSABI_NSK       OSABI = 14 /* HP Non-Stop Kerne
134     ELFOSABI_ARM       OSABI = 97 /* ARM */
135     ELFOSABI_STANDALONE OSABI = 255 /* Standalone (embed
136 )
137
138 var osabiStrings = []intName{
139     {0, "ELFOSABI_NONE"},
140     {1, "ELFOSABI_HPUX"},
141     {2, "ELFOSABI_NETBSD"},
142     {3, "ELFOSABI_LINUX"},
143     {4, "ELFOSABI_HURD"},

```

```

144     {5, "ELFOSABI_86OPEN"},
145     {6, "ELFOSABI_SOLARIS"},
146     {7, "ELFOSABI_AIX"},
147     {8, "ELFOSABI_IRIX"},
148     {9, "ELFOSABI_FREEBSD"},
149     {10, "ELFOSABI_TRU64"},
150     {11, "ELFOSABI_MODESTO"},
151     {12, "ELFOSABI_OPENBSD"},
152     {13, "ELFOSABI_OPENVMS"},
153     {14, "ELFOSABI_NSK"},
154     {97, "ELFOSABI_ARM"},
155     {255, "ELFOSABI_STANDALONE"},
156 }
157
158 func (i OSABI) String() string { return stringName(uint32(i)) }
159 func (i OSABI) GoString() string { return stringName(uint32(i)) }
160
161 // Type is found in Header.Type.
162 type Type uint16
163
164 const (
165     ET_NONE    Type = 0        /* Unknown type. */
166     ET_REL     Type = 1        /* Relocatable. */
167     ET_EXEC    Type = 2        /* Executable. */
168     ET_DYN     Type = 3        /* Shared object. */
169     ET_CORE    Type = 4        /* Core file. */
170     ET_LOOS    Type = 0xfe00   /* First operating system sp
171     ET_HIOS    Type = 0xfeff   /* Last operating system-spe
172     ET_LOPROC  Type = 0xff00   /* First processor-specific.
173     ET_HIPROC  Type = 0xffff   /* Last processor-specific.
174 )
175
176 var typeStrings = []intName{
177     {0, "ET_NONE"},
178     {1, "ET_REL"},
179     {2, "ET_EXEC"},
180     {3, "ET_DYN"},
181     {4, "ET_CORE"},
182     {0xfe00, "ET_LOOS"},
183     {0xfeff, "ET_HIOS"},
184     {0xff00, "ET_LOPROC"},
185     {0xffff, "ET_HIPROC"},
186 }
187
188 func (i Type) String() string { return stringName(uint32(i)) }
189 func (i Type) GoString() string { return stringName(uint32(i)) }
190
191 // Machine is found in Header.Machine.
192 type Machine uint16

```

```

193
194 const (
195     EM_NONE           Machine = 0 /* Unknown machine. */
196     EM_M32            Machine = 1 /* AT&T WE32100. */
197     EM_SPARC          Machine = 2 /* Sun SPARC. */
198     EM_386            Machine = 3 /* Intel i386. */
199     EM_68K            Machine = 4 /* Motorola 68000. */
200     EM_88K            Machine = 5 /* Motorola 88000. */
201     EM_860            Machine = 7 /* Intel i860. */
202     EM_MIPS           Machine = 8 /* MIPS R3000 Big-Endian
203     EM_S370           Machine = 9 /* IBM System/370. */
204     EM_MIPS_RS3_LE    Machine = 10 /* MIPS R3000 Little-End
205     EM_PARISC         Machine = 15 /* HP PA-RISC. */
206     EM_VPP500        Machine = 17 /* Fujitsu VPP500. */
207     EM_SPARC32PLUS    Machine = 18 /* SPARC v8plus. */
208     EM_960            Machine = 19 /* Intel 80960. */
209     EM_PPC            Machine = 20 /* PowerPC 32-bit. */
210     EM_PPC64          Machine = 21 /* PowerPC 64-bit. */
211     EM_S390           Machine = 22 /* IBM System/390. */
212     EM_V800           Machine = 36 /* NEC V800. */
213     EM_FR20           Machine = 37 /* Fujitsu FR20. */
214     EM_RH32           Machine = 38 /* TRW RH-32. */
215     EM_RCE            Machine = 39 /* Motorola RCE. */
216     EM_ARM            Machine = 40 /* ARM. */
217     EM_SH             Machine = 42 /* Hitachi SH. */
218     EM_SPARCV9        Machine = 43 /* SPARC v9 64-bit. */
219     EM_TRICORE        Machine = 44 /* Siemens TriCore embed
220     EM_ARC            Machine = 45 /* Argonaut RISC Core. *
221     EM_H8_300         Machine = 46 /* Hitachi H8/300. */
222     EM_H8_300H        Machine = 47 /* Hitachi H8/300H. */
223     EM_H8S            Machine = 48 /* Hitachi H8S. */
224     EM_H8_500         Machine = 49 /* Hitachi H8/500. */
225     EM_IA_64          Machine = 50 /* Intel IA-64 Processor
226     EM_MIPS_X         Machine = 51 /* Stanford MIPS-X. */
227     EM_COLDFIRE       Machine = 52 /* Motorola ColdFire. */
228     EM_68HC12         Machine = 53 /* Motorola M68HC12. */
229     EM_MMA            Machine = 54 /* Fujitsu MMA. */
230     EM_PCP            Machine = 55 /* Siemens PCP. */
231     EM_NCPU           Machine = 56 /* Sony nCPU. */
232     EM_NDR1           Machine = 57 /* Denso NDR1 microproce
233     EM_STARCORE       Machine = 58 /* Motorola Star*Core pr
234     EM_ME16           Machine = 59 /* Toyota ME16 processor
235     EM_ST100          Machine = 60 /* STMicroelectronics ST
236     EM_TINYJ          Machine = 61 /* Advanced Logic Corp.
237     EM_X86_64         Machine = 62 /* Advanced Micro Device
238
239     /* Non-standard or deprecated. */
240     EM_486            Machine = 6 /* Intel i486. */
241     EM_MIPS_RS4_BE    Machine = 10 /* MIPS R4000 Big-En
242     EM_ALPHA_STD      Machine = 41 /* Digital Alpha (st

```

```

243         EM_ALPHA           Machine = 0x9026 /* Alpha (written in
244     )
245
246     var machineStrings = []intName{
247         {0, "EM_NONE"},
248         {1, "EM_M32"},
249         {2, "EM_SPARC"},
250         {3, "EM_386"},
251         {4, "EM_68K"},
252         {5, "EM_88K"},
253         {7, "EM_860"},
254         {8, "EM_MIPS"},
255         {9, "EM_S370"},
256         {10, "EM_MIPS_RS3_LE"},
257         {15, "EM_PARISC"},
258         {17, "EM_VPP500"},
259         {18, "EM_SPARC32PLUS"},
260         {19, "EM_960"},
261         {20, "EM_PPC"},
262         {21, "EM_PPC64"},
263         {22, "EM_S390"},
264         {36, "EM_V800"},
265         {37, "EM_FR20"},
266         {38, "EM_RH32"},
267         {39, "EM_RCE"},
268         {40, "EM_ARM"},
269         {42, "EM_SH"},
270         {43, "EM_SPARCV9"},
271         {44, "EM_TRICORE"},
272         {45, "EM_ARC"},
273         {46, "EM_H8_300"},
274         {47, "EM_H8_300H"},
275         {48, "EM_H8S"},
276         {49, "EM_H8_500"},
277         {50, "EM_IA_64"},
278         {51, "EM_MIPS_X"},
279         {52, "EM_COLDFIRE"},
280         {53, "EM_68HC12"},
281         {54, "EM_MMA"},
282         {55, "EM_PCP"},
283         {56, "EM_NCPU"},
284         {57, "EM_NDR1"},
285         {58, "EM_STARCORE"},
286         {59, "EM_ME16"},
287         {60, "EM_ST100"},
288         {61, "EM_TINYJ"},
289         {62, "EM_X86_64"},
290
291         /* Non-standard or deprecated. */

```

```

292         {6, "EM_486"},
293         {10, "EM_MIPS_RS4_BE"},
294         {41, "EM_ALPHA_STD"},
295         {0x9026, "EM_ALPHA"},
296     }
297
298     func (i Machine) String() string { return stringName(uint32(i)) }
299     func (i Machine) GoString() string { return stringName(uint32(i)) }
300
301     // Special section indices.
302     type SectionIndex int
303
304     const (
305         SHN_UNDEF      SectionIndex = 0          /* Undefined, mi
306         SHN_LORESERVE  SectionIndex = 0xff00     /* First of rese
307         SHN_LOPROC     SectionIndex = 0xff00     /* First process
308         SHN_HIPROC     SectionIndex = 0xff1f     /* Last processo
309         SHN_LOOS       SectionIndex = 0xff20     /* First operati
310         SHN_HIOS       SectionIndex = 0xff3f     /* Last operatin
311         SHN_ABS        SectionIndex = 0xffff1    /* Absolute valu
312         SHN_COMMON     SectionIndex = 0xffff2    /* Common data.
313         SHN_XINDEX     SectionIndex = 0xffff    /* Escape -- ind
314         SHN_HIRESERVE  SectionIndex = 0xffff    /* Last of reser
315     )
316
317     var shnStrings = []intName{
318         {0, "SHN_UNDEF"},
319         {0xff00, "SHN_LOPROC"},
320         {0xff20, "SHN_LOOS"},
321         {0xffff1, "SHN_ABS"},
322         {0xffff2, "SHN_COMMON"},
323         {0xffff, "SHN_XINDEX"},
324     }
325
326     func (i SectionIndex) String() string { return stringName(int(i)) }
327     func (i SectionIndex) GoString() string { return stringName(int(i)) }
328
329     // Section type.
330     type SectionType uint32
331
332     const (
333         SHT_NULL          SectionType = 0          /* inact
334         SHT_PROGBITS     SectionType = 1          /* progr
335         SHT_SYMTAB       SectionType = 2          /* symbo
336         SHT_STRTAB       SectionType = 3          /* strin
337         SHT_RELA         SectionType = 4          /* reloc
338         SHT_HASH         SectionType = 5          /* symbo
339         SHT_DYNAMIC      SectionType = 6          /* dynam
340         SHT_NOTE         SectionType = 7          /* note

```

```

341     SHT_NOBITS           SectionType = 8           /* no sp
342     SHT_REL              SectionType = 9           /* reloc
343     SHT_SHLIB           SectionType = 10          /* reser
344     SHT_DYNSYM          SectionType = 11          /* dynam
345     SHT_INIT_ARRAY      SectionType = 14          /* Initi
346     SHT_FINI_ARRAY      SectionType = 15          /* Termi
347     SHT_PREINIT_ARRAY   SectionType = 16          /* Pre-i
348     SHT_GROUP           SectionType = 17          /* Secti
349     SHT_SYMTAB_SHNDX    SectionType = 18          /* Secti
350     SHT_LOOS            SectionType = 0x60000000 /* First
351     SHT_GNU_ATTRIBUTES  SectionType = 0x6fffffff5 /* GNU o
352     SHT_GNU_HASH        SectionType = 0x6fffffff6 /* GNU h
353     SHT_GNU_LIBLIST     SectionType = 0x6fffffff7 /* GNU p
354     SHT_GNU_VERDEF      SectionType = 0x6fffffffD /* GNU v
355     SHT_GNU_VERNEED     SectionType = 0x6fffffffE /* GNU v
356     SHT_GNU_VERSYM      SectionType = 0x6fffffffF /* GNU v
357     SHT_HIOS            SectionType = 0x6fffffffF /* Last
358     SHT_LOPROC          SectionType = 0x70000000 /* reser
359     SHT_HIPROC          SectionType = 0x7fffffffF /* speci
360     SHT_LOUSER          SectionType = 0x80000000 /* reser
361     SHT_HIUSER          SectionType = 0xffffffff /* speci

```

```

362 )

```

```

363
364 var shtStrings = []intName{
365     {0, "SHT_NULL"},
366     {1, "SHT_PROGBITS"},
367     {2, "SHT_SYMTAB"},
368     {3, "SHT_STRTAB"},
369     {4, "SHT_RELA"},
370     {5, "SHT_HASH"},
371     {6, "SHT_DYNAMIC"},
372     {7, "SHT_NOTE"},
373     {8, "SHT_NOBITS"},
374     {9, "SHT_REL"},
375     {10, "SHT_SHLIB"},
376     {11, "SHT_DYNSYM"},
377     {14, "SHT_INIT_ARRAY"},
378     {15, "SHT_FINI_ARRAY"},
379     {16, "SHT_PREINIT_ARRAY"},
380     {17, "SHT_GROUP"},
381     {18, "SHT_SYMTAB_SHNDX"},
382     {0x60000000, "SHT_LOOS"},
383     {0x6fffffff5, "SHT_GNU_ATTRIBUTES"},
384     {0x6fffffff6, "SHT_GNU_HASH"},
385     {0x6fffffff7, "SHT_GNU_LIBLIST"},
386     {0x6fffffffD, "SHT_GNU_VERDEF"},
387     {0x6fffffffE, "SHT_GNU_VERNEED"},
388     {0x6fffffffF, "SHT_GNU_VERSYM"},
389     {0x70000000, "SHT_LOPROC"},
390     {0x7fffffffF, "SHT_HIPROC"},

```

```

391         {0x80000000, "SHT_LOUSER"},
392         {0xffffffff, "SHT_HIUSER"},
393     }
394
395     func (i SectionType) String() string { return stringName(u
396     func (i SectionType) GoString() string { return stringName(u
397
398     // Section flags.
399     type SectionFlag uint32
400
401     const (
402         SHF_WRITE      SectionFlag = 0x1      /* Sec
403         SHF_ALLOC      SectionFlag = 0x2      /* Sec
404         SHF_EXECINSTR  SectionFlag = 0x4      /* Sec
405         SHF_MERGE      SectionFlag = 0x10     /* Sec
406         SHF_STRINGS    SectionFlag = 0x20     /* Sec
407         SHF_INFO_LINK  SectionFlag = 0x40     /* sh_
408         SHF_LINK_ORDER SectionFlag = 0x80     /* Spe
409         SHF_OS_NONCONFORMING SectionFlag = 0x100 /* OS-
410         SHF_GROUP      SectionFlag = 0x200    /* Merr
411         SHF_TLS        SectionFlag = 0x400    /* Sec
412         SHF_MASKOS     SectionFlag = 0x0ff00000 /* OS-
413         SHF_MASKPROC   SectionFlag = 0xf0000000 /* Pro
414     )
415
416     var shfStrings = []intName{
417         {0x1, "SHF_WRITE"},
418         {0x2, "SHF_ALLOC"},
419         {0x4, "SHF_EXECINSTR"},
420         {0x10, "SHF_MERGE"},
421         {0x20, "SHF_STRINGS"},
422         {0x40, "SHF_INFO_LINK"},
423         {0x80, "SHF_LINK_ORDER"},
424         {0x100, "SHF_OS_NONCONFORMING"},
425         {0x200, "SHF_GROUP"},
426         {0x400, "SHF_TLS"},
427     }
428
429     func (i SectionFlag) String() string { return flagName(uin
430     func (i SectionFlag) GoString() string { return flagName(uin
431
432     // Prog.Type
433     type ProgType int
434
435     const (
436         PT_NULL      ProgType = 0      /* Unused entry. */
437         PT_LOAD      ProgType = 1      /* Loadable segment
438         PT_DYNAMIC   ProgType = 2      /* Dynamic linking
439         PT_INTERP    ProgType = 3      /* Pathname of inte

```

```

440         PT_NOTE      ProgType = 4          /* Auxiliary inform
441         PT_SHLIB     ProgType = 5          /* Reserved (not us
442         PT_PHDR     ProgType = 6          /* Location of prog
443         PT_TLS      ProgType = 7          /* Thread local sto
444         PT_LOOS     ProgType = 0x60000000 /* First OS-specifi
445         PT_HIOS     ProgType = 0x6fffffff /* Last OS-specific
446         PT_LOPROC   ProgType = 0x70000000 /* First processor-
447         PT_HIPROC   ProgType = 0x7fffffff /* Last processor-s
448     )
449
450     var ptStrings = []intName{
451         {0, "PT_NULL"},
452         {1, "PT_LOAD"},
453         {2, "PT_DYNAMIC"},
454         {3, "PT_INTERP"},
455         {4, "PT_NOTE"},
456         {5, "PT_SHLIB"},
457         {6, "PT_PHDR"},
458         {7, "PT_TLS"},
459         {0x60000000, "PT_LOOS"},
460         {0x6fffffff, "PT_HIOS"},
461         {0x70000000, "PT_LOPROC"},
462         {0x7fffffff, "PT_HIPROC"},
463     }
464
465     func (i ProgType) String() string { return stringName(uint
466     func (i ProgType) GoString() string { return stringName(uint
467
468     // Prog.Flag
469     type ProgFlag uint32
470
471     const (
472         PF_X          ProgFlag = 0x1          /* Executable. */
473         PF_W          ProgFlag = 0x2          /* Writable. */
474         PF_R          ProgFlag = 0x4          /* Readable. */
475         PF_MASKOS    ProgFlag = 0x0ff00000 /* Operating syste
476         PF_MASKPROC  ProgFlag = 0xf0000000 /* Processor-speci
477     )
478
479     var pfStrings = []intName{
480         {0x1, "PF_X"},
481         {0x2, "PF_W"},
482         {0x4, "PF_R"},
483     }
484
485     func (i ProgFlag) String() string { return flagName(uint32
486     func (i ProgFlag) GoString() string { return flagName(uint32
487
488     // Dyn.Tag

```

```

489 type DynTag int
490
491 const (
492     DT_NULL           DynTag = 0 /* Terminating entry. */
493     DT_NEEDED         DynTag = 1 /* String table offset o
494     DT_PLTRELSZ       DynTag = 2 /* Total size in bytes o
495     DT_PLTGOT         DynTag = 3 /* Processor-dependent a
496     DT_HASH           DynTag = 4 /* Address of symbol has
497     DT_STRTAB         DynTag = 5 /* Address of string tab
498     DT_SYMTAB         DynTag = 6 /* Address of symbol tab
499     DT_RELA           DynTag = 7 /* Address of ElfNN_Rela
500     DT_RELASZ         DynTag = 8 /* Total size of ElfNN_R
501     DT_RELAENT        DynTag = 9 /* Size of each ElfNN_Re
502     DT_STRSZ          DynTag = 10 /* Size of string table.
503     DT_SYMENT         DynTag = 11 /* Size of each symbol t
504     DT_INIT           DynTag = 12 /* Address of initializa
505     DT_FINI           DynTag = 13 /* Address of finalizati
506     DT_SONAME         DynTag = 14 /* String table offset o
507     DT_RPATH          DynTag = 15 /* String table offset o
508     DT_SYMBOLIC       DynTag = 16 /* Indicates "symbolic"
509     DT_REL            DynTag = 17 /* Address of ElfNN_Rel
510     DT_RELSZ         DynTag = 18 /* Total size of ElfNN_R
511     DT_RELENT        DynTag = 19 /* Size of each ElfNN_Re
512     DT_PLTREL         DynTag = 20 /* Type of relocation us
513     DT_DEBUG          DynTag = 21 /* Reserved (not used).
514     DT_TEXTREL        DynTag = 22 /* Indicates there may b
515     DT_JMPREL         DynTag = 23 /* Address of PLT reloca
516     DT_BIND_NOW       DynTag = 24 /* [sup] */
517     DT_INIT_ARRAY     DynTag = 25 /* Address of the array
518     DT_FINI_ARRAY     DynTag = 26 /* Address of the array
519     DT_INIT_ARRAYSZ   DynTag = 27 /* Size in bytes of the
520     DT_FINI_ARRAYSZ   DynTag = 28 /* Size in bytes of the
521     DT_RUNPATH        DynTag = 29 /* String table offset o
522     DT_FLAGS          DynTag = 30 /* Object specific flag
523     DT_ENCODING       DynTag = 32 /* Values greater than o
524         and less than DT_LOOS follow the rules for
525         the interpretation of the d_un union
526         as follows: even == 'd_ptr', even == 'd_val'
527         or none */
528     DT_PREINIT_ARRAY  DynTag = 32 /* Address of
529     DT_PREINIT_ARRAYSZ DynTag = 33 /* Size in by
530     DT_LOOS           DynTag = 0x6000000d /* First OS-s
531     DT_HIOS           DynTag = 0x6ffff000 /* Last OS-sp
532     DT_VERSYM         DynTag = 0x6fffffff0
533     DT_VERNEED        DynTag = 0x6fffffffef
534     DT_VERNEEDNUM     DynTag = 0x6fffffff
535     DT_LOPROC         DynTag = 0x70000000 /* First proc
536     DT_HIPROC         DynTag = 0x7fffffff /* Last proce
537 )
538

```

```

539 var dtStrings = []intName{
540     {0, "DT_NULL"},
541     {1, "DT_NEEDED"},
542     {2, "DT_PLTRELSZ"},
543     {3, "DT_PLTGOT"},
544     {4, "DT_HASH"},
545     {5, "DT_STRTAB"},
546     {6, "DT_SYMTAB"},
547     {7, "DT_RELA"},
548     {8, "DT_RELASZ"},
549     {9, "DT_RELAENT"},
550     {10, "DT_STRSZ"},
551     {11, "DT_SYMENT"},
552     {12, "DT_INIT"},
553     {13, "DT_FINI"},
554     {14, "DT_SONAME"},
555     {15, "DT_RPATH"},
556     {16, "DT_SYMBOLIC"},
557     {17, "DT_REL"},
558     {18, "DT_RELSZ"},
559     {19, "DT_RELENT"},
560     {20, "DT_PLTREL"},
561     {21, "DT_DEBUG"},
562     {22, "DT_TEXTREL"},
563     {23, "DT_JMPREL"},
564     {24, "DT_BIND_NOW"},
565     {25, "DT_INIT_ARRAY"},
566     {26, "DT_FINI_ARRAY"},
567     {27, "DT_INIT_ARRAYSZ"},
568     {28, "DT_FINI_ARRAYSZ"},
569     {29, "DT_RUNPATH"},
570     {30, "DT_FLAGS"},
571     {32, "DT_ENCODING"},
572     {32, "DT_PREINIT_ARRAY"},
573     {33, "DT_PREINIT_ARRAYSZ"},
574     {0x6000000d, "DT_LOOS"},
575     {0x6ffff000, "DT_HIOS"},
576     {0x6fffffff0, "DT_VERSYM"},
577     {0x6fffffffef, "DT_VERNEED"},
578     {0x6fffffff, "DT_VERNEEDNUM"},
579     {0x70000000, "DT_LOPROC"},
580     {0x7fffffff, "DT_HIPROC"},
581 }
582
583 func (i DynTag) String() string { return stringName(uint32(i)) }
584 func (i DynTag) GoString() string { return stringName(uint32(i)) }
585
586 // DT_FLAGS values.
587 type DynFlag int

```

```

588
589 const (
590     DF_ORIGIN DynFlag = 0x0001 /* Indicates that the obj
591         make reference to the
592         $ORIGIN substitution string */
593     DF_SYMBOLIC DynFlag = 0x0002 /* Indicates "symbolic"
594     DF_TEXTREL  DynFlag = 0x0004 /* Indicates there may
595     DF_BIND_NOW DynFlag = 0x0008 /* Indicates that the d
596         process all relocations for the object
597         containing this entry before transferring
598         control to the program. */
599     DF_STATIC_TLS DynFlag = 0x0010 /* Indicates that the
600         executable contains code using a static
601         thread-local storage scheme. */
602 )
603
604 var dflagStrings = []intName{
605     {0x0001, "DF_ORIGIN"},
606     {0x0002, "DF_SYMBOLIC"},
607     {0x0004, "DF_TEXTREL"},
608     {0x0008, "DF_BIND_NOW"},
609     {0x0010, "DF_STATIC_TLS"},
610 }
611
612 func (i DynFlag) String() string { return flagName(uint32(i)) }
613 func (i DynFlag) GoString() string { return flagName(uint32(i)) }
614
615 // NType values; used in core files.
616 type NType int
617
618 const (
619     NT_PRSTATUS NType = 1 /* Process status. */
620     NT_FPREGSET NType = 2 /* Floating point registers. */
621     NT_PRPSINFO NType = 3 /* Process state info. */
622 )
623
624 var ntypeStrings = []intName{
625     {1, "NT_PRSTATUS"},
626     {2, "NT_FPREGSET"},
627     {3, "NT_PRPSINFO"},
628 }
629
630 func (i NType) String() string { return stringName(uint32(i)) }
631 func (i NType) GoString() string { return stringName(uint32(i)) }
632
633 /* Symbol Binding - ELFNN_ST_BIND - st_info */
634 type SymBind int
635
636 const (

```

```

637         STB_LOCAL   SymBind = 0 /* Local symbol */
638         STB_GLOBAL  SymBind = 1 /* Global symbol */
639         STB_WEAK    SymBind = 2 /* like global - lower prece
640         STB_LOOS    SymBind = 10 /* Reserved range for operat
641         STB_HIOS    SymBind = 12 /* specific semantics. */
642         STB_LOPROC  SymBind = 13 /* reserved range for proces
643         STB_HIPROC  SymBind = 15 /* specific semantics. */
644     )
645
646     var stbStrings = []intName{
647         {0, "STB_LOCAL"},
648         {1, "STB_GLOBAL"},
649         {2, "STB_WEAK"},
650         {10, "STB_LOOS"},
651         {12, "STB_HIOS"},
652         {13, "STB_LOPROC"},
653         {15, "STB_HIPROC"},
654     }
655
656     func (i SymBind) String() string {return stringName(uint3
657     func (i SymBind) GoString() string {return stringName(uint3
658
659     /* Symbol type - ELFNN_ST_TYPE - st_info */
660     type SymType int
661
662     const (
663         STT_NOTYPE   SymType = 0 /* Unspecified type. */
664         STT_OBJECT   SymType = 1 /* Data object. */
665         STT_FUNC     SymType = 2 /* Function. */
666         STT_SECTION  SymType = 3 /* Section. */
667         STT_FILE     SymType = 4 /* Source file. */
668         STT_COMMON   SymType = 5 /* Uninitialized common blo
669         STT_TLS      SymType = 6 /* TLS object. */
670         STT_LOOS     SymType = 10 /* Reserved range for opera
671         STT_HIOS     SymType = 12 /* specific semantics. */
672         STT_LOPROC   SymType = 13 /* reserved range for proce
673         STT_HIPROC   SymType = 15 /* specific semantics. */
674     )
675
676     var sttStrings = []intName{
677         {0, "STT_NOTYPE"},
678         {1, "STT_OBJECT"},
679         {2, "STT_FUNC"},
680         {3, "STT_SECTION"},
681         {4, "STT_FILE"},
682         {5, "STT_COMMON"},
683         {6, "STT_TLS"},
684         {10, "STT_LOOS"},
685         {12, "STT_HIOS"},
686         {13, "STT_LOPROC"},

```

```

687         {15, "STT_HIPROC"},
688     }
689
690     func (i SymType) String() string { return stringName(uint32(i)) }
691     func (i SymType) GoString() string { return stringName(uint32(i)) }
692
693     /* Symbol visibility - ELFNN_ST_VISIBILITY - st_other */
694     type SymVis int
695
696     const (
697         STV_DEFAULT    SymVis = 0x0 /* Default visibility (see STT_HIPROC) */
698         STV_INTERNAL   SymVis = 0x1 /* Special meaning in relocation */
699         STV_HIDDEN     SymVis = 0x2 /* Not visible. */
700         STV_PROTECTED  SymVis = 0x3 /* Visible but not preemptible */
701     )
702
703     var stvStrings = []intName{
704         {0x0, "STV_DEFAULT"},
705         {0x1, "STV_INTERNAL"},
706         {0x2, "STV_HIDDEN"},
707         {0x3, "STV_PROTECTED"},
708     }
709
710     func (i SymVis) String() string { return stringName(uint32(i)) }
711     func (i SymVis) GoString() string { return stringName(uint32(i)) }
712
713     /*
714      * Relocation types.
715      */
716
717     // Relocation types for x86-64.
718     type R_X86_64 int
719
720     const (
721         R_X86_64_NONE      R_X86_64 = 0 /* No relocation. */
722         R_X86_64_64       R_X86_64 = 1 /* Add 64 bit symbol */
723         R_X86_64_PC32     R_X86_64 = 2 /* PC-relative 32 bit */
724         R_X86_64_GOT32    R_X86_64 = 3 /* PC-relative 32 bit */
725         R_X86_64_PLT32    R_X86_64 = 4 /* PC-relative 32 bit */
726         R_X86_64_COPY     R_X86_64 = 5 /* Copy data from symbol */
727         R_X86_64_GLOB_DAT R_X86_64 = 6 /* Set GOT entry to symbol */
728         R_X86_64_JMP_SLOT R_X86_64 = 7 /* Set GOT entry to symbol */
729         R_X86_64_RELATIVE R_X86_64 = 8 /* Add load address of symbol */
730         R_X86_64_GOTPCREL R_X86_64 = 9 /* Add 32 bit signed symbol */
731         R_X86_64_32       R_X86_64 = 10 /* Add 32 bit zero extended symbol */
732         R_X86_64_32S     R_X86_64 = 11 /* Add 32 bit sign extended symbol */
733         R_X86_64_16      R_X86_64 = 12 /* Add 16 bit zero extended symbol */
734         R_X86_64_PC16    R_X86_64 = 13 /* Add 16 bit signed symbol */
735         R_X86_64_8       R_X86_64 = 14 /* Add 8 bit zero extended symbol */

```

```

736         R_X86_64_PC8           R_X86_64 = 15 /* Add 8 bit signed
737         R_X86_64_DTPMOD64      R_X86_64 = 16 /* ID of module cont
738         R_X86_64_DTPOFF64     R_X86_64 = 17 /* Offset in TLS blo
739         R_X86_64_TPOFF64     R_X86_64 = 18 /* Offset in static
740         R_X86_64_TLSD         R_X86_64 = 19 /* PC relative offse
741         R_X86_64_TLSD         R_X86_64 = 20 /* PC relative offse
742         R_X86_64_DTPOFF32     R_X86_64 = 21 /* Offset in TLS blo
743         R_X86_64_GOTTPOFF     R_X86_64 = 22 /* PC relative offse
744         R_X86_64_TPOFF32     R_X86_64 = 23 /* Offset in static
745     )
746
747     var rx86_64Strings = []intName{
748         {0, "R_X86_64_NONE"},
749         {1, "R_X86_64_64"},
750         {2, "R_X86_64_PC32"},
751         {3, "R_X86_64_GOT32"},
752         {4, "R_X86_64_PLT32"},
753         {5, "R_X86_64_COPY"},
754         {6, "R_X86_64_GLOB_DAT"},
755         {7, "R_X86_64_JMP_SLOT"},
756         {8, "R_X86_64_RELATIVE"},
757         {9, "R_X86_64_GOTPCREL"},
758         {10, "R_X86_64_32"},
759         {11, "R_X86_64_32S"},
760         {12, "R_X86_64_16"},
761         {13, "R_X86_64_PC16"},
762         {14, "R_X86_64_8"},
763         {15, "R_X86_64_PC8"},
764         {16, "R_X86_64_DTPMOD64"},
765         {17, "R_X86_64_DTPOFF64"},
766         {18, "R_X86_64_TPOFF64"},
767         {19, "R_X86_64_TLSD"},
768         {20, "R_X86_64_TLSD"},
769         {21, "R_X86_64_DTPOFF32"},
770         {22, "R_X86_64_GOTTPOFF"},
771         {23, "R_X86_64_TPOFF32"},
772     }
773
774     func (i R_X86_64) String() string { return stringName(uint(i)) }
775     func (i R_X86_64) GoString() string { return stringName(uint(i)) }
776
777     // Relocation types for Alpha.
778     type R_ALPHA int
779
780     const (
781         R_ALPHA_NONE           R_ALPHA = 0 /* No reloc */
782         R_ALPHA_REFLONG        R_ALPHA = 1 /* Direct 32 bit
783         R_ALPHA_REFQUAD       R_ALPHA = 2 /* Direct 64 bit
784         R_ALPHA_GPREL32       R_ALPHA = 3 /* GP relative 3

```

```

785     R_ALPHA_LITERAL           R_ALPHA = 4 /* GP relative 1
786     R_ALPHA_LITUSE           R_ALPHA = 5 /* Optimization
787     R_ALPHA_GPDISP          R_ALPHA = 6 /* Add displacem
788     R_ALPHA_BRADDR          R_ALPHA = 7 /* PC+4 relative
789     R_ALPHA_HINT             R_ALPHA = 8 /* PC+4 relative
790     R_ALPHA_SREL16           R_ALPHA = 9 /* PC relative 1
791     R_ALPHA_SREL32           R_ALPHA = 10 /* PC relative 3
792     R_ALPHA_SREL64           R_ALPHA = 11 /* PC relative 6
793     R_ALPHA_OP_PUSH          R_ALPHA = 12 /* OP stack push
794     R_ALPHA_OP_STORE         R_ALPHA = 13 /* OP stack pop
795     R_ALPHA_OP_PSUB          R_ALPHA = 14 /* OP stack subt
796     R_ALPHA_OP_PRSHIFT       R_ALPHA = 15 /* OP stack righ
797     R_ALPHA_GPVALUE          R_ALPHA = 16
798     R_ALPHA_GPRELHIGH        R_ALPHA = 17
799     R_ALPHA_GPRELLOW         R_ALPHA = 18
800     R_ALPHA_IMMED_GP_16      R_ALPHA = 19
801     R_ALPHA_IMMED_GP_HI32    R_ALPHA = 20
802     R_ALPHA_IMMED_SCN_HI32   R_ALPHA = 21
803     R_ALPHA_IMMED_BR_HI32    R_ALPHA = 22
804     R_ALPHA_IMMED_L032       R_ALPHA = 23
805     R_ALPHA_COPY             R_ALPHA = 24 /* Copy symbol a
806     R_ALPHA_GLOB_DAT         R_ALPHA = 25 /* Create GOT en
807     R_ALPHA_JMP_SLOT         R_ALPHA = 26 /* Create PLT en
808     R_ALPHA_RELATIVE         R_ALPHA = 27 /* Adjust by pro

```

```

809 )

```

```

810
811 var ralphaStrings = []intName{
812     {0, "R_ALPHA_NONE"},
813     {1, "R_ALPHA_REFLONG"},
814     {2, "R_ALPHA_REFQUAD"},
815     {3, "R_ALPHA_GPREL32"},
816     {4, "R_ALPHA_LITERAL"},
817     {5, "R_ALPHA_LITUSE"},
818     {6, "R_ALPHA_GPDISP"},
819     {7, "R_ALPHA_BRADDR"},
820     {8, "R_ALPHA_HINT"},
821     {9, "R_ALPHA_SREL16"},
822     {10, "R_ALPHA_SREL32"},
823     {11, "R_ALPHA_SREL64"},
824     {12, "R_ALPHA_OP_PUSH"},
825     {13, "R_ALPHA_OP_STORE"},
826     {14, "R_ALPHA_OP_PSUB"},
827     {15, "R_ALPHA_OP_PRSHIFT"},
828     {16, "R_ALPHA_GPVALUE"},
829     {17, "R_ALPHA_GPRELHIGH"},
830     {18, "R_ALPHA_GPRELLOW"},
831     {19, "R_ALPHA_IMMED_GP_16"},
832     {20, "R_ALPHA_IMMED_GP_HI32"},
833     {21, "R_ALPHA_IMMED_SCN_HI32"},
834     {22, "R_ALPHA_IMMED_BR_HI32"},

```

```

835     {23, "R_ALPHA_IMMED_L032"},
836     {24, "R_ALPHA_COPY"},
837     {25, "R_ALPHA_GLOB_DAT"},
838     {26, "R_ALPHA_JMP_SLOT"},
839     {27, "R_ALPHA_RELATIVE"},
840 }
841
842 func (i R_ALPHA) String() string { return stringName(uint32(i)) }
843 func (i R_ALPHA) GoString() string { return stringName(uint32(i)) }
844
845 // Relocation types for ARM.
846 type R_ARM int
847
848 const (
849     R_ARM_NONE          R_ARM = 0 /* No relocation. */
850     R_ARM_PC24          R_ARM = 1
851     R_ARM_ABS32         R_ARM = 2
852     R_ARM_REL32         R_ARM = 3
853     R_ARM_PC13          R_ARM = 4
854     R_ARM_ABS16         R_ARM = 5
855     R_ARM_ABS12         R_ARM = 6
856     R_ARM_THM_ABS5     R_ARM = 7
857     R_ARM_ABS8         R_ARM = 8
858     R_ARM_SBREL32      R_ARM = 9
859     R_ARM_THM_PC22     R_ARM = 10
860     R_ARM_THM_PC8      R_ARM = 11
861     R_ARM_ARMV8_VCALL9 R_ARM = 12
862     R_ARM_SWI24        R_ARM = 13
863     R_ARM_THM_SWI8     R_ARM = 14
864     R_ARM_XPC25        R_ARM = 15
865     R_ARM_THM_XPC22    R_ARM = 16
866     R_ARM_COPY         R_ARM = 20 /* Copy data from sha
867     R_ARM_GLOB_DAT     R_ARM = 21 /* Set GOT entry to d
868     R_ARM_JUMP_SLOT    R_ARM = 22 /* Set GOT entry to c
869     R_ARM_RELATIVE     R_ARM = 23 /* Add load address c
870     R_ARM_GOTOFF       R_ARM = 24 /* Add GOT-relative s
871     R_ARM_GOTPC        R_ARM = 25 /* Add PC-relative GC
872     R_ARM_GOT32        R_ARM = 26 /* Add PC-relative GC
873     R_ARM_PLT32        R_ARM = 27 /* Add PC-relative PL
874     R_ARM_GNU_VTENTRY  R_ARM = 100
875     R_ARM_GNU_VTINHERIT R_ARM = 101
876     R_ARM_RSBREL32     R_ARM = 250
877     R_ARM_THM_RPC22    R_ARM = 251
878     R_ARM_RREL32       R_ARM = 252
879     R_ARM_RABS32       R_ARM = 253
880     R_ARM_RPC24        R_ARM = 254
881     R_ARM_RBASE        R_ARM = 255
882 )
883

```

```

884 var rarmStrings = []intName{
885     {0, "R_ARM_NONE"},
886     {1, "R_ARM_PC24"},
887     {2, "R_ARM_ABS32"},
888     {3, "R_ARM_REL32"},
889     {4, "R_ARM_PC13"},
890     {5, "R_ARM_ABS16"},
891     {6, "R_ARM_ABS12"},
892     {7, "R_ARM_THM_ABS5"},
893     {8, "R_ARM_ABS8"},
894     {9, "R_ARM_SBREL32"},
895     {10, "R_ARM_THM_PC22"},
896     {11, "R_ARM_THM_PC8"},
897     {12, "R_ARM_AMP_VCALL9"},
898     {13, "R_ARM_SWI24"},
899     {14, "R_ARM_THM_SWI8"},
900     {15, "R_ARM_XPC25"},
901     {16, "R_ARM_THM_XPC22"},
902     {20, "R_ARM_COPY"},
903     {21, "R_ARM_GLOB_DAT"},
904     {22, "R_ARM_JUMP_SLOT"},
905     {23, "R_ARM_RELATIVE"},
906     {24, "R_ARM_GOTOFF"},
907     {25, "R_ARM_GOTPC"},
908     {26, "R_ARM_GOT32"},
909     {27, "R_ARM_PLT32"},
910     {100, "R_ARM_GNU_VTENTRY"},
911     {101, "R_ARM_GNU_VTINHERIT"},
912     {250, "R_ARM_RSBREL32"},
913     {251, "R_ARM_THM_RPC22"},
914     {252, "R_ARM_RREL32"},
915     {253, "R_ARM_RABS32"},
916     {254, "R_ARM_RPC24"},
917     {255, "R_ARM_RBASE"},
918 }
919
920 func (i R_ARM) String() string { return stringName(uint32(i)) }
921 func (i R_ARM) GoString() string { return stringName(uint32(i)) }
922
923 // Relocation types for 386.
924 type R_386 int
925
926 const (
927     R_386_NONE      R_386 = 0 /* No relocation. */
928     R_386_32        R_386 = 1 /* Add symbol value. */
929     R_386_PC32      R_386 = 2 /* Add PC-relative sym
930     R_386_GOT32     R_386 = 3 /* Add PC-relative GOT
931     R_386_PLT32     R_386 = 4 /* Add PC-relative PLT
932     R_386_COPY      R_386 = 5 /* Copy data from shar

```

```

933     R_386_GLOB_DAT      R_386 = 6 /* Set GOT entry to da
934     R_386_JMP_SLOT      R_386 = 7 /* Set GOT entry to co
935     R_386_RELATIVE      R_386 = 8 /* Add load address of
936     R_386_GOTOFF        R_386 = 9 /* Add GOT-relative sy
937     R_386_GOTPC         R_386 = 10 /* Add PC-relative GOT
938     R_386_TLS_TPOFF     R_386 = 14 /* Negative offset in
939     R_386_TLS_IE        R_386 = 15 /* Absolute address of
940     R_386_TLS_GOTIE     R_386 = 16 /* GOT entry for negat
941     R_386_TLS_LE        R_386 = 17 /* Negative offset rel
942     R_386_TLS_GD        R_386 = 18 /* 32 bit offset to GC
943     R_386_TLS_LDM       R_386 = 19 /* 32 bit offset to GC
944     R_386_TLS_GD_32     R_386 = 24 /* 32 bit offset to GC
945     R_386_TLS_GD_PUSH   R_386 = 25 /* pushl instruction f
946     R_386_TLS_GD_CALL   R_386 = 26 /* call instruction fo
947     R_386_TLS_GD_POP    R_386 = 27 /* popl instruction fo
948     R_386_TLS_LDM_32    R_386 = 28 /* 32 bit offset to GC
949     R_386_TLS_LDM_PUSH  R_386 = 29 /* pushl instruction f
950     R_386_TLS_LDM_CALL  R_386 = 30 /* call instruction fo
951     R_386_TLS_LDM_POP   R_386 = 31 /* popl instruction fo
952     R_386_TLS_LDO_32    R_386 = 32 /* 32 bit offset from
953     R_386_TLS_IE_32     R_386 = 33 /* 32 bit offset to GC
954     R_386_TLS_LE_32     R_386 = 34 /* 32 bit offset withi
955     R_386_TLS_DTPMOD32  R_386 = 35 /* GOT entry containin
956     R_386_TLS_DTPOFF32  R_386 = 36 /* GOT entry containin
957     R_386_TLS_TPOFF32   R_386 = 37 /* GOT entry of -ve st
958 )
959
960 var r386Strings = []intName{
961     {0, "R_386_NONE"},
962     {1, "R_386_32"},
963     {2, "R_386_PC32"},
964     {3, "R_386_GOT32"},
965     {4, "R_386_PLT32"},
966     {5, "R_386_COPY"},
967     {6, "R_386_GLOB_DAT"},
968     {7, "R_386_JMP_SLOT"},
969     {8, "R_386_RELATIVE"},
970     {9, "R_386_GOTOFF"},
971     {10, "R_386_GOTPC"},
972     {14, "R_386_TLS_TPOFF"},
973     {15, "R_386_TLS_IE"},
974     {16, "R_386_TLS_GOTIE"},
975     {17, "R_386_TLS_LE"},
976     {18, "R_386_TLS_GD"},
977     {19, "R_386_TLS_LDM"},
978     {24, "R_386_TLS_GD_32"},
979     {25, "R_386_TLS_GD_PUSH"},
980     {26, "R_386_TLS_GD_CALL"},
981     {27, "R_386_TLS_GD_POP"},
982     {28, "R_386_TLS_LDM_32"},

```

```

983     {29, "R_386_TLS_LDM_PUSH"},
984     {30, "R_386_TLS_LDM_CALL"},
985     {31, "R_386_TLS_LDM_POP"},
986     {32, "R_386_TLS_LDO_32"},
987     {33, "R_386_TLS_IE_32"},
988     {34, "R_386_TLS_LE_32"},
989     {35, "R_386_TLS_DTPMOD32"},
990     {36, "R_386_TLS_DTPOFF32"},
991     {37, "R_386_TLS_TPOFF32"},
992 }
993
994 func (i R_386) String() string { return stringName(uint32(i)) }
995 func (i R_386) GoString() string { return stringName(uint32(i)) }
996
997 // Relocation types for PowerPC.
998 type R_PPC int
999
1000 const (
1001     R_PPC_NONE          R_PPC = 0 /* No relocation. */
1002     R_PPC_ADDR32        R_PPC = 1
1003     R_PPC_ADDR24        R_PPC = 2
1004     R_PPC_ADDR16        R_PPC = 3
1005     R_PPC_ADDR16_LO     R_PPC = 4
1006     R_PPC_ADDR16_HI     R_PPC = 5
1007     R_PPC_ADDR16_HA     R_PPC = 6
1008     R_PPC_ADDR14        R_PPC = 7
1009     R_PPC_ADDR14_BRTAKEN R_PPC = 8
1010     R_PPC_ADDR14_BRNTAKEN R_PPC = 9
1011     R_PPC_REL24         R_PPC = 10
1012     R_PPC_REL14         R_PPC = 11
1013     R_PPC_REL14_BRTAKEN R_PPC = 12
1014     R_PPC_REL14_BRNTAKEN R_PPC = 13
1015     R_PPC_GOT16         R_PPC = 14
1016     R_PPC_GOT16_LO     R_PPC = 15
1017     R_PPC_GOT16_HI     R_PPC = 16
1018     R_PPC_GOT16_HA     R_PPC = 17
1019     R_PPC_PLTREL24     R_PPC = 18
1020     R_PPC_COPY          R_PPC = 19
1021     R_PPC_GLOB_DAT     R_PPC = 20
1022     R_PPC_JMP_SLOT     R_PPC = 21
1023     R_PPC_RELATIVE     R_PPC = 22
1024     R_PPC_LOCAL24PC    R_PPC = 23
1025     R_PPC_UADDR32      R_PPC = 24
1026     R_PPC_UADDR16      R_PPC = 25
1027     R_PPC_REL32        R_PPC = 26
1028     R_PPC_PLT32        R_PPC = 27
1029     R_PPC_PLTREL32    R_PPC = 28
1030     R_PPC_PLT16_LO     R_PPC = 29
1031     R_PPC_PLT16_HI     R_PPC = 30

```

```
1032         R_PPC_PLT16_HA           R_PPC = 31
1033         R_PPC_SDAREL16           R_PPC = 32
1034         R_PPC_SECTOFF           R_PPC = 33
1035         R_PPC_SECTOFF_LO        R_PPC = 34
1036         R_PPC_SECTOFF_HI        R_PPC = 35
1037         R_PPC_SECTOFF_HA        R_PPC = 36
1038         R_PPC_TLS                R_PPC = 67
1039         R_PPC_DTPMOD32           R_PPC = 68
1040         R_PPC_TPREL16           R_PPC = 69
1041         R_PPC_TPREL16_LO        R_PPC = 70
1042         R_PPC_TPREL16_HI        R_PPC = 71
1043         R_PPC_TPREL16_HA        R_PPC = 72
1044         R_PPC_TPREL32           R_PPC = 73
1045         R_PPC_DTPREL16           R_PPC = 74
1046         R_PPC_DTPREL16_LO      R_PPC = 75
1047         R_PPC_DTPREL16_HI      R_PPC = 76
1048         R_PPC_DTPREL16_HA      R_PPC = 77
1049         R_PPC_DTPREL32         R_PPC = 78
1050         R_PPC_GOT_TLSGD16       R_PPC = 79
1051         R_PPC_GOT_TLSGD16_LO    R_PPC = 80
1052         R_PPC_GOT_TLSGD16_HI    R_PPC = 81
1053         R_PPC_GOT_TLSGD16_HA    R_PPC = 82
1054         R_PPC_GOT_TLSLD16       R_PPC = 83
1055         R_PPC_GOT_TLSLD16_LO    R_PPC = 84
1056         R_PPC_GOT_TLSLD16_HI    R_PPC = 85
1057         R_PPC_GOT_TLSLD16_HA    R_PPC = 86
1058         R_PPC_GOT_TPREL16       R_PPC = 87
1059         R_PPC_GOT_TPREL16_LO    R_PPC = 88
1060         R_PPC_GOT_TPREL16_HI    R_PPC = 89
1061         R_PPC_GOT_TPREL16_HA    R_PPC = 90
1062         R_PPC_EMB_NADDR32       R_PPC = 101
1063         R_PPC_EMB_NADDR16       R_PPC = 102
1064         R_PPC_EMB_NADDR16_LO    R_PPC = 103
1065         R_PPC_EMB_NADDR16_HI    R_PPC = 104
1066         R_PPC_EMB_NADDR16_HA    R_PPC = 105
1067         R_PPC_EMB_SDAI16        R_PPC = 106
1068         R_PPC_EMB_SDA2I16       R_PPC = 107
1069         R_PPC_EMB_SDA2REL       R_PPC = 108
1070         R_PPC_EMB_SDA21         R_PPC = 109
1071         R_PPC_EMB_MRKREF        R_PPC = 110
1072         R_PPC_EMB_RELSEC16      R_PPC = 111
1073         R_PPC_EMB_RELST_LO      R_PPC = 112
1074         R_PPC_EMB_RELST_HI      R_PPC = 113
1075         R_PPC_EMB_RELST_HA      R_PPC = 114
1076         R_PPC_EMB_BIT_FLD       R_PPC = 115
1077         R_PPC_EMB_RELSDA        R_PPC = 116
1078     )
1079
1080     var rppcStrings = []intName{
```

1081 {0, "R_PPC_NONE"},
1082 {1, "R_PPC_ADDR32"},
1083 {2, "R_PPC_ADDR24"},
1084 {3, "R_PPC_ADDR16"},
1085 {4, "R_PPC_ADDR16_LO"},
1086 {5, "R_PPC_ADDR16_HI"},
1087 {6, "R_PPC_ADDR16_HA"},
1088 {7, "R_PPC_ADDR14"},
1089 {8, "R_PPC_ADDR14_BRTAKEN"},
1090 {9, "R_PPC_ADDR14_BRNTAKEN"},
1091 {10, "R_PPC_REL24"},
1092 {11, "R_PPC_REL14"},
1093 {12, "R_PPC_REL14_BRTAKEN"},
1094 {13, "R_PPC_REL14_BRNTAKEN"},
1095 {14, "R_PPC_GOT16"},
1096 {15, "R_PPC_GOT16_LO"},
1097 {16, "R_PPC_GOT16_HI"},
1098 {17, "R_PPC_GOT16_HA"},
1099 {18, "R_PPC_PLTREL24"},
1100 {19, "R_PPC_COPY"},
1101 {20, "R_PPC_GLOB_DAT"},
1102 {21, "R_PPC_JMP_SLOT"},
1103 {22, "R_PPC_RELATIVE"},
1104 {23, "R_PPC_LOCAL24PC"},
1105 {24, "R_PPC_UADDR32"},
1106 {25, "R_PPC_UADDR16"},
1107 {26, "R_PPC_REL32"},
1108 {27, "R_PPC_PLT32"},
1109 {28, "R_PPC_PLTREL32"},
1110 {29, "R_PPC_PLT16_LO"},
1111 {30, "R_PPC_PLT16_HI"},
1112 {31, "R_PPC_PLT16_HA"},
1113 {32, "R_PPC_SDAREL16"},
1114 {33, "R_PPC_SECTOFF"},
1115 {34, "R_PPC_SECTOFF_LO"},
1116 {35, "R_PPC_SECTOFF_HI"},
1117 {36, "R_PPC_SECTOFF_HA"},
1118
1119 {67, "R_PPC_TLS"},
1120 {68, "R_PPC_DTPMOD32"},
1121 {69, "R_PPC_TPREL16"},
1122 {70, "R_PPC_TPREL16_LO"},
1123 {71, "R_PPC_TPREL16_HI"},
1124 {72, "R_PPC_TPREL16_HA"},
1125 {73, "R_PPC_TPREL32"},
1126 {74, "R_PPC_DTPREL16"},
1127 {75, "R_PPC_DTPREL16_LO"},
1128 {76, "R_PPC_DTPREL16_HI"},
1129 {77, "R_PPC_DTPREL16_HA"},
1130 {78, "R_PPC_DTPREL32"},

```

1131     {79, "R_PPC_GOT_TLSGD16"},
1132     {80, "R_PPC_GOT_TLSGD16_LO"},
1133     {81, "R_PPC_GOT_TLSGD16_HI"},
1134     {82, "R_PPC_GOT_TLSGD16_HA"},
1135     {83, "R_PPC_GOT_TLSLD16"},
1136     {84, "R_PPC_GOT_TLSLD16_LO"},
1137     {85, "R_PPC_GOT_TLSLD16_HI"},
1138     {86, "R_PPC_GOT_TLSLD16_HA"},
1139     {87, "R_PPC_GOT_TPREL16"},
1140     {88, "R_PPC_GOT_TPREL16_LO"},
1141     {89, "R_PPC_GOT_TPREL16_HI"},
1142     {90, "R_PPC_GOT_TPREL16_HA"},
1143
1144     {101, "R_PPC_EMB_NADDR32"},
1145     {102, "R_PPC_EMB_NADDR16"},
1146     {103, "R_PPC_EMB_NADDR16_LO"},
1147     {104, "R_PPC_EMB_NADDR16_HI"},
1148     {105, "R_PPC_EMB_NADDR16_HA"},
1149     {106, "R_PPC_EMB_SDAI16"},
1150     {107, "R_PPC_EMB_SDA2I16"},
1151     {108, "R_PPC_EMB_SDA2REL"},
1152     {109, "R_PPC_EMB_SDA21"},
1153     {110, "R_PPC_EMB_MRKREF"},
1154     {111, "R_PPC_EMB_RELSEC16"},
1155     {112, "R_PPC_EMB_RELST_LO"},
1156     {113, "R_PPC_EMB_RELST_HI"},
1157     {114, "R_PPC_EMB_RELST_HA"},
1158     {115, "R_PPC_EMB_BIT_FLD"},
1159     {116, "R_PPC_EMB_RELSDA"},
1160 }
1161
1162 func (i R_PPC) String() string { return stringName(uint32(
1163 func (i R_PPC) GoString() string { return stringName(uint32(
1164
1165 // Relocation types for SPARC.
1166 type R_SPARC int
1167
1168 const (
1169     R_SPARC_NONE      R_SPARC = 0
1170     R_SPARC_8         R_SPARC = 1
1171     R_SPARC_16        R_SPARC = 2
1172     R_SPARC_32        R_SPARC = 3
1173     R_SPARC_DISP8     R_SPARC = 4
1174     R_SPARC_DISP16    R_SPARC = 5
1175     R_SPARC_DISP32    R_SPARC = 6
1176     R_SPARC_WDISP30   R_SPARC = 7
1177     R_SPARC_WDISP22   R_SPARC = 8
1178     R_SPARC_HI22      R_SPARC = 9
1179     R_SPARC_22        R_SPARC = 10

```

```

1180         R_SPARC_13           R_SPARC = 11
1181         R_SPARC_LO10         R_SPARC = 12
1182         R_SPARC_GOT10        R_SPARC = 13
1183         R_SPARC_GOT13        R_SPARC = 14
1184         R_SPARC_GOT22        R_SPARC = 15
1185         R_SPARC_PC10          R_SPARC = 16
1186         R_SPARC_PC22          R_SPARC = 17
1187         R_SPARC_WPLT30        R_SPARC = 18
1188         R_SPARC_COPY          R_SPARC = 19
1189         R_SPARC_GLOB_DAT      R_SPARC = 20
1190         R_SPARC_JMP_SLOT      R_SPARC = 21
1191         R_SPARC_RELATIVE      R_SPARC = 22
1192         R_SPARC_UA32          R_SPARC = 23
1193         R_SPARC_PLT32         R_SPARC = 24
1194         R_SPARC_HIPLT22       R_SPARC = 25
1195         R_SPARC_LOPLT10       R_SPARC = 26
1196         R_SPARC_PCPLT32       R_SPARC = 27
1197         R_SPARC_PCPLT22       R_SPARC = 28
1198         R_SPARC_PCPLT10       R_SPARC = 29
1199         R_SPARC_10            R_SPARC = 30
1200         R_SPARC_11            R_SPARC = 31
1201         R_SPARC_64            R_SPARC = 32
1202         R_SPARC_OL010         R_SPARC = 33
1203         R_SPARC_HH22          R_SPARC = 34
1204         R_SPARC_HM10          R_SPARC = 35
1205         R_SPARC_LM22          R_SPARC = 36
1206         R_SPARC_PC_HH22       R_SPARC = 37
1207         R_SPARC_PC_HM10       R_SPARC = 38
1208         R_SPARC_PC_LM22       R_SPARC = 39
1209         R_SPARC_WDISP16       R_SPARC = 40
1210         R_SPARC_WDISP19       R_SPARC = 41
1211         R_SPARC_GLOB_JMP      R_SPARC = 42
1212         R_SPARC_7             R_SPARC = 43
1213         R_SPARC_5             R_SPARC = 44
1214         R_SPARC_6             R_SPARC = 45
1215         R_SPARC_DISP64        R_SPARC = 46
1216         R_SPARC_PLT64         R_SPARC = 47
1217         R_SPARC_HIX22         R_SPARC = 48
1218         R_SPARC_LOX10         R_SPARC = 49
1219         R_SPARC_H44           R_SPARC = 50
1220         R_SPARC_M44           R_SPARC = 51
1221         R_SPARC_L44           R_SPARC = 52
1222         R_SPARC_REGISTER      R_SPARC = 53
1223         R_SPARC_UA64          R_SPARC = 54
1224         R_SPARC_UA16          R_SPARC = 55
1225     )
1226
1227     var rsparcStrings = []intName{
1228         {0, "R_SPARC_NONE"},

```

1229 {1, "R_SPARC_8"},
1230 {2, "R_SPARC_16"},
1231 {3, "R_SPARC_32"},
1232 {4, "R_SPARC_DISP8"},
1233 {5, "R_SPARC_DISP16"},
1234 {6, "R_SPARC_DISP32"},
1235 {7, "R_SPARC_WDISP30"},
1236 {8, "R_SPARC_WDISP22"},
1237 {9, "R_SPARC_HI22"},
1238 {10, "R_SPARC_22"},
1239 {11, "R_SPARC_13"},
1240 {12, "R_SPARC_LO10"},
1241 {13, "R_SPARC_GOT10"},
1242 {14, "R_SPARC_GOT13"},
1243 {15, "R_SPARC_GOT22"},
1244 {16, "R_SPARC_PC10"},
1245 {17, "R_SPARC_PC22"},
1246 {18, "R_SPARC_WPLT30"},
1247 {19, "R_SPARC_COPY"},
1248 {20, "R_SPARC_GLOB_DAT"},
1249 {21, "R_SPARC_JMP_SLOT"},
1250 {22, "R_SPARC_RELATIVE"},
1251 {23, "R_SPARC_UA32"},
1252 {24, "R_SPARC_PLT32"},
1253 {25, "R_SPARC_HIPLT22"},
1254 {26, "R_SPARC_LOPLT10"},
1255 {27, "R_SPARC_PCPLT32"},
1256 {28, "R_SPARC_PCPLT22"},
1257 {29, "R_SPARC_PCPLT10"},
1258 {30, "R_SPARC_10"},
1259 {31, "R_SPARC_11"},
1260 {32, "R_SPARC_64"},
1261 {33, "R_SPARC_OLO10"},
1262 {34, "R_SPARC_HH22"},
1263 {35, "R_SPARC_HM10"},
1264 {36, "R_SPARC_LM22"},
1265 {37, "R_SPARC_PC_HH22"},
1266 {38, "R_SPARC_PC_HM10"},
1267 {39, "R_SPARC_PC_LM22"},
1268 {40, "R_SPARC_WDISP16"},
1269 {41, "R_SPARC_WDISP19"},
1270 {42, "R_SPARC_GLOB_JMP"},
1271 {43, "R_SPARC_7"},
1272 {44, "R_SPARC_5"},
1273 {45, "R_SPARC_6"},
1274 {46, "R_SPARC_DISP64"},
1275 {47, "R_SPARC_PLT64"},
1276 {48, "R_SPARC_HIX22"},
1277 {49, "R_SPARC_LOX10"},
1278 {50, "R_SPARC_H44"},

```

1279         {51, "R_SPARC_M44"},
1280         {52, "R_SPARC_L44"},
1281         {53, "R_SPARC_REGISTER"},
1282         {54, "R_SPARC_UA64"},
1283         {55, "R_SPARC_UA16"},
1284     }
1285
1286     func (i R_SPARC) String() string { return stringName(uint32(i)) }
1287     func (i R_SPARC) GoString() string { return stringName(uint32(i)) }
1288
1289     // Magic number for the elf trampoline, chosen wisely to be
1290     const ARM_MAGIC_TRAMP_NUMBER = 0x5c000003
1291
1292     // ELF32 File header.
1293     type Header32 struct {
1294         Ident    [EI_NIDENT]byte /* File identification. */
1295         Type     uint16          /* File type. */
1296         Machine  uint16          /* Machine architecture. */
1297         Version  uint32          /* ELF format version. */
1298         Entry    uint32          /* Entry point. */
1299         Phoff    uint32          /* Program header file off
1300         Shoff    uint32          /* Section header file off
1301         Flags    uint32          /* Architecture-specific f
1302         Ehsize   uint16          /* Size of ELF header in b
1303         Phentsize uint16        /* Size of program header
1304         Phnum    uint16          /* Number of program heade
1305         Shentsize uint16        /* Size of section header
1306         Shnum    uint16          /* Number of section heade
1307         Shstrndx uint16          /* Section name strings se
1308     }
1309
1310     // ELF32 Section header.
1311     type Section32 struct {
1312         Name     uint32 /* Section name (index into the sec
1313         Type     uint32 /* Section type. */
1314         Flags    uint32 /* Section flags. */
1315         Addr     uint32 /* Address in memory image. */
1316         Off      uint32 /* Offset in file. */
1317         Size     uint32 /* Size in bytes. */
1318         Link     uint32 /* Index of a related section. */
1319         Info     uint32 /* Depends on section type. */
1320         Addralign uint32 /* Alignment in bytes. */
1321         Entsize  uint32 /* Size of each entry in section. */
1322     }
1323
1324     // ELF32 Program header.
1325     type Prog32 struct {
1326         Type     uint32 /* Entry type. */
1327         Off      uint32 /* File offset of contents. */

```

```

1328         Vaddr  uint32 /* Virtual address in memory image. */
1329         Paddr  uint32 /* Physical address (not used). */
1330         Filesz uint32 /* Size of contents in file. */
1331         Memsz  uint32 /* Size of contents in memory. */
1332         Flags  uint32 /* Access permission flags. */
1333         Align  uint32 /* Alignment in memory and file. */
1334     }
1335
1336 // ELF32 Dynamic structure. The ".dynamic" section contains
1337 type Dyn32 struct {
1338     Tag int32 /* Entry type. */
1339     Val uint32 /* Integer/Address value. */
1340 }
1341
1342 /*
1343  * Relocation entries.
1344  */
1345
1346 // ELF32 Relocations that don't need an addend field.
1347 type Rel32 struct {
1348     Off  uint32 /* Location to be relocated. */
1349     Info uint32 /* Relocation type and symbol index. */
1350 }
1351
1352 // ELF32 Relocations that need an addend field.
1353 type Rela32 struct {
1354     Off      uint32 /* Location to be relocated. */
1355     Info     uint32 /* Relocation type and symbol index. */
1356     Addend   int32 /* Addend. */
1357 }
1358
1359 func R_SYM32(info uint32) uint32      { return uint32(info >
1360 func R_TYPE32(info uint32) uint32    { return uint32(info &
1361 func R_INF032(sym, typ uint32) uint32 { return sym<<8 | typ
1362
1363 // ELF32 Symbol.
1364 type Sym32 struct {
1365     Name  uint32
1366     Value uint32
1367     Size  uint32
1368     Info  uint8
1369     Other uint8
1370     Shndx uint16
1371 }
1372
1373 const Sym32Size = 16
1374
1375 func ST_BIND(info uint8) SymBind { return SymBind(info >> 4)
1376 func ST_TYPE(info uint8) SymType { return SymType(info & 0xF

```

```

1377 func ST_INFO(bind SymBind, typ SymType) uint8 {
1378     return uint8(bind)<<4 | uint8(typ)&0xf
1379 }
1380 func ST_VISIBILITY(other uint8) SymVis { return SymVis(other
1381
1382 /*
1383  * ELF64
1384  */
1385
1386 // ELF64 file header.
1387 type Header64 struct {
1388     Ident    [EI_NIDENT]byte /* File identification. */
1389     Type     uint16          /* File type. */
1390     Machine  uint16          /* Machine architecture. */
1391     Version  uint32          /* ELF format version. */
1392     Entry    uint64          /* Entry point. */
1393     Phoff    uint64          /* Program header file off
1394     Shoff    uint64          /* Section header file off
1395     Flags    uint32          /* Architecture-specific f
1396     Ehsize   uint16          /* Size of ELF header in b
1397     Phentsize uint16         /* Size of program header
1398     Phnum    uint16          /* Number of program heade
1399     Shentsize uint16         /* Size of section header
1400     Shnum    uint16          /* Number of section heade
1401     Shstrndx uint16          /* Section name strings se
1402 }
1403
1404 // ELF64 Section header.
1405 type Section64 struct {
1406     Name     uint32 /* Section name (index into the sec
1407     Type     uint32 /* Section type. */
1408     Flags    uint64 /* Section flags. */
1409     Addr     uint64 /* Address in memory image. */
1410     Off      uint64 /* Offset in file. */
1411     Size     uint64 /* Size in bytes. */
1412     Link     uint32 /* Index of a related section. */
1413     Info     uint32 /* Depends on section type. */
1414     Addralign uint64 /* Alignment in bytes. */
1415     Entsize  uint64 /* Size of each entry in section. */
1416 }
1417
1418 // ELF64 Program header.
1419 type Prog64 struct {
1420     Type     uint32 /* Entry type. */
1421     Flags    uint32 /* Access permission flags. */
1422     Off      uint64 /* File offset of contents. */
1423     Vaddr    uint64 /* Virtual address in memory image. */
1424     Paddr    uint64 /* Physical address (not used). */
1425     Filesz   uint64 /* Size of contents in file. */
1426     Memsz    uint64 /* Size of contents in memory. */

```

```

1427         Align uint64 /* Alignment in memory and file. */
1428     }
1429
1430 // ELF64 Dynamic structure. The ".dynamic" section contains
1431 type Dyn64 struct {
1432     Tag int64 /* Entry type. */
1433     Val uint64 /* Integer/address value */
1434 }
1435
1436 /*
1437  * Relocation entries.
1438  */
1439
1440 /* ELF64 relocations that don't need an addend field. */
1441 type Rel64 struct {
1442     Off uint64 /* Location to be relocated. */
1443     Info uint64 /* Relocation type and symbol index. */
1444 }
1445
1446 /* ELF64 relocations that need an addend field. */
1447 type Rela64 struct {
1448     Off uint64 /* Location to be relocated. */
1449     Info uint64 /* Relocation type and symbol index. */
1450     Addend int64 /* Addend. */
1451 }
1452
1453 func R_SYM64(info uint64) uint32 { return uint32(info >>
1454 func R_TYPE64(info uint64) uint32 { return uint32(info) }
1455 func R_INFO(sym, typ uint32) uint64 { return uint64(sym)<<32
1456
1457 // ELF64 symbol table entries.
1458 type Sym64 struct {
1459     Name uint32 /* String table index of name. */
1460     Info uint8 /* Type and binding information. */
1461     Other uint8 /* Reserved (not used). */
1462     Shndx uint16 /* Section index of symbol. */
1463     Value uint64 /* Symbol value. */
1464     Size uint64 /* Size of associated object. */
1465 }
1466
1467 const Sym64Size = 24
1468
1469 type intName struct {
1470     i uint32
1471     s string
1472 }
1473
1474 func stringName(i uint32, names []intName, goSyntax bool) st
1475     for _, n := range names {

```

```

1476         if n.i == i {
1477             if goSyntax {
1478                 return "elf." + n.s
1479             }
1480             return n.s
1481         }
1482     }
1483
1484     // second pass - look for smaller to add with.
1485     // assume sorted already
1486     for j := len(names) - 1; j >= 0; j-- {
1487         n := names[j]
1488         if n.i < i {
1489             s := n.s
1490             if goSyntax {
1491                 s = "elf." + s
1492             }
1493             return s + "+" + strconv.FormatUint(
1494         }
1495     }
1496
1497     return strconv.FormatUint(uint64(i), 10)
1498 }
1499
1500 func flagName(i uint32, names []intName, goSyntax bool) stri
1501     s := ""
1502     for _, n := range names {
1503         if n.i&i == n.i {
1504             if len(s) > 0 {
1505                 s += "+"
1506             }
1507             if goSyntax {
1508                 s += "elf."
1509             }
1510             s += n.s
1511             i -= n.i
1512         }
1513     }
1514     if len(s) == 0 {
1515         return "0x" + strconv.FormatUint(uint64(i),
1516     }
1517     if i != 0 {
1518         s += "+0x" + strconv.FormatUint(uint64(i), 1
1519     }
1520     return s
1521 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/debug/elf/file.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package elf implements access to ELF object files.
6 package elf
7
8 import (
9     "bytes"
10    "debug/dwarf"
11    "encoding/binary"
12    "errors"
13    "fmt"
14    "io"
15    "os"
16 )
17
18 // TODO: error reporting detail
19
20 /*
21  * Internal ELF representation
22  */
23
24 // A FileHeader represents an ELF file header.
25 type FileHeader struct {
26     Class      Class
27     Data       Data
28     Version    Version
29     OSABI      OSABI
30     ABIVersion uint8
31     ByteOrder  binary.ByteOrder
32     Type       Type
33     Machine    Machine
34 }
35
36 // A File represents an open ELF file.
37 type File struct {
38     FileHeader
39     Sections []*Section
40     Progs    []*Prog
41     closer   io.Closer
42     gnuNeed  []verneed
43     gnuVersym []byte
44 }
```

```

45
46 // A SectionHeader represents a single ELF section header.
47 type SectionHeader struct {
48     Name      string
49     Type      SectionType
50     Flags     SectionFlag
51     Addr      uint64
52     Offset    uint64
53     Size      uint64
54     Link      uint32
55     Info      uint32
56     Addralign uint64
57     Entsize   uint64
58 }
59
60 // A Section represents a single section in an ELF file.
61 type Section struct {
62     SectionHeader
63
64     // Embed ReaderAt for ReadAt method.
65     // Do not embed SectionReader directly
66     // to avoid having Read and Seek.
67     // If a client wants Read and Seek it must use
68     // Open() to avoid fighting over the seek offset
69     // with other clients.
70     io.ReaderAt
71     sr *io.SectionReader
72 }
73
74 // Data reads and returns the contents of the ELF section.
75 func (s *Section) Data() ([]byte, error) {
76     dat := make([]byte, s.sr.Size())
77     n, err := s.sr.ReadAt(dat, 0)
78     return dat[0:n], err
79 }
80
81 // stringTable reads and returns the string table given by t
82 // specified link value.
83 func (f *File) stringTable(link uint32) ([]byte, error) {
84     if link <= 0 || link >= uint32(len(f.Sections)) {
85         return nil, errors.New("section has invalid
86     }
87     return f.Sections[link].Data()
88 }
89
90 // Open returns a new ReadSeeker reading the ELF section.
91 func (s *Section) Open() io.ReadSeeker { return io.NewSectionReader(s.sr, s.sr.Size())
92
93 // A ProgHeader represents a single ELF program header.
94 type ProgHeader struct {

```

```

95         Type    ProgType
96         Flags   ProgFlag
97         Off     uint64
98         Vaddr   uint64
99         Paddr   uint64
100        Filesz  uint64
101        Memsz   uint64
102        Align   uint64
103    }
104
105    // A Prog represents a single ELF program header in an ELF b
106    type Prog struct {
107        ProgHeader
108
109        // Embed ReaderAt for ReadAt method.
110        // Do not embed SectionReader directly
111        // to avoid having Read and Seek.
112        // If a client wants Read and Seek it must use
113        // Open() to avoid fighting over the seek offset
114        // with other clients.
115        io.ReaderAt
116        sr *io.SectionReader
117    }
118
119    // Open returns a new ReadSeeker reading the ELF program bod
120    func (p *Prog) Open() io.ReadSeeker { return io.NewSectionRe
121
122    // A Symbol represents an entry in an ELF symbol table secti
123    type Symbol struct {
124        Name        string
125        Info, Other byte
126        Section     SectionIndex
127        Value, Size uint64
128    }
129
130    /*
131     * ELF reader
132     */
133
134    type FormatError struct {
135        off int64
136        msg string
137        val interface{}}
138    }
139
140    func (e *FormatError) Error() string {
141        msg := e.msg
142        if e.val != nil {
143            msg += fmt.Sprintf(" '%v' ", e.val)

```

```

144     }
145     msg += fmt.Sprintf("in record at byte %#x", e.off)
146     return msg
147 }
148
149 // Open opens the named file using os.Open and prepares it f
150 func Open(name string) (*File, error) {
151     f, err := os.Open(name)
152     if err != nil {
153         return nil, err
154     }
155     ff, err := NewFile(f)
156     if err != nil {
157         f.Close()
158         return nil, err
159     }
160     ff.closer = f
161     return ff, nil
162 }
163
164 // Close closes the File.
165 // If the File was created using NewFile directly instead of
166 // Close has no effect.
167 func (f *File) Close() error {
168     var err error
169     if f.closer != nil {
170         err = f.closer.Close()
171         f.closer = nil
172     }
173     return err
174 }
175
176 // SectionByType returns the first section in f with the
177 // given type, or nil if there is no such section.
178 func (f *File) SectionByType(typ SectionType) *Section {
179     for _, s := range f.Sections {
180         if s.Type == typ {
181             return s
182         }
183     }
184     return nil
185 }
186
187 // NewFile creates a new File for accessing an ELF binary in
188 // The ELF binary is expected to start at position 0 in the
189 func NewFile(r io.ReaderAt) (*File, error) {
190     sr := io.NewSectionReader(r, 0, 1<<63-1)
191     // Read and decode ELF identifier
192     var ident [16]uint8

```

```

193     if _, err := r.ReadAt(ident[0:], 0); err != nil {
194         return nil, err
195     }
196     if ident[0] != '\x7f' || ident[1] != 'E' || ident[2]
197         return nil, &FormatError{0, "bad magic numbe
198     }
199
200     f := new(File)
201     f.Class = Class(ident[EI_CLASS])
202     switch f.Class {
203     case ELFCLASS32:
204     case ELFCLASS64:
205         // ok
206     default:
207         return nil, &FormatError{0, "unknown ELF cla
208     }
209
210     f.Data = Data(ident[EI_DATA])
211     switch f.Data {
212     case ELFDATA2LSB:
213         f.ByteOrder = binary.LittleEndian
214     case ELFDATA2MSB:
215         f.ByteOrder = binary.BigEndian
216     default:
217         return nil, &FormatError{0, "unknown ELF dat
218     }
219
220     f.Version = Version(ident[EI_VERSION])
221     if f.Version != EV_CURRENT {
222         return nil, &FormatError{0, "unknown ELF ver
223     }
224
225     f.OSABI = OSABI(ident[EI_OSABI])
226     f.ABIVersion = ident[EI_ABIVERSION]
227
228     // Read ELF file header
229     var phoff int64
230     var phentsize, phnum int
231     var shoff int64
232     var shentsize, shnum, shstrndx int
233     shstrndx = -1
234     switch f.Class {
235     case ELFCLASS32:
236         hdr := new(Header32)
237         sr.Seek(0, os.SEEK_SET)
238         if err := binary.Read(sr, f.ByteOrder, hdr);
239             return nil, err
240     }
241     f.Type = Type(hdr.Type)
242     f.Machine = Machine(hdr.Machine)

```

```

243         if v := Version(hdr.Version); v != f.Version
244             return nil, &FormatError{0, "mismatch"}
245     }
246     phoff = int64(hdr.Phoff)
247     phentsize = int(hdr.Phentsize)
248     phnum = int(hdr.Phnum)
249     shoff = int64(hdr.Shoff)
250     shentsize = int(hdr.Shentsize)
251     shnum = int(hdr.Shnum)
252     shstrndx = int(hdr.Shstrndx)
253     case ELFCLASS64:
254         hdr := new(Header64)
255         sr.Seek(0, os.SEEK_SET)
256         if err := binary.Read(sr, f.ByteOrder, hdr);
257             return nil, err
258     }
259     f.Type = Type(hdr.Type)
260     f.Machine = Machine(hdr.Machine)
261     if v := Version(hdr.Version); v != f.Version
262         return nil, &FormatError{0, "mismatch"}
263     }
264     phoff = int64(hdr.Phoff)
265     phentsize = int(hdr.Phentsize)
266     phnum = int(hdr.Phnum)
267     shoff = int64(hdr.Shoff)
268     shentsize = int(hdr.Shentsize)
269     shnum = int(hdr.Shnum)
270     shstrndx = int(hdr.Shstrndx)
271 }
272 if shstrndx < 0 || shstrndx >= shnum {
273     return nil, &FormatError{0, "invalid ELF shs"}
274 }
275
276 // Read program headers
277 f.Progs = make([]*Prog, phnum)
278 for i := 0; i < phnum; i++ {
279     off := phoff + int64(i)*int64(phentsize)
280     sr.Seek(off, os.SEEK_SET)
281     p := new(Prog)
282     switch f.Class {
283     case ELFCLASS32:
284         ph := new(Prog32)
285         if err := binary.Read(sr, f.ByteOrder, ph);
286             return nil, err
287     }
288     p.ProgHeader = ProgHeader{
289         Type: ProgType(ph.Type),
290         Flags: ProgFlag(ph.Flags),
291         Off: uint64(ph.Off),

```

```

292             Vaddr: uint64(ph.Vaddr),
293             Paddr: uint64(ph.Paddr),
294             Filesz: uint64(ph.FileSZ),
295             MemSZ: uint64(ph.MemSZ),
296             Align: uint64(ph.Align),
297         }
298     case ELFCLASS64:
299         ph := new(Prog64)
300         if err := binary.Read(sr, f.ByteOrder)
301             return nil, err
302     }
303     p.ProgHeader = ProgHeader{
304         Type: ProgType(ph.Type),
305         Flags: ProgFlag(ph.Flags),
306         Off: uint64(ph.Off),
307         Vaddr: uint64(ph.Vaddr),
308         Paddr: uint64(ph.Paddr),
309         Filesz: uint64(ph.FileSZ),
310         MemSZ: uint64(ph.MemSZ),
311         Align: uint64(ph.Align),
312     }
313 }
314 p.sr = io.NewSectionReader(r, int64(p.Off),
315 p.ReaderAt = p.sr
316 f.Progs[i] = p
317 }
318
319 // Read section headers
320 f.Sections = make([]*Section, shnum)
321 names := make([]uint32, shnum)
322 for i := 0; i < shnum; i++ {
323     off := shoff + int64(i)*int64(shentsize)
324     sr.Seek(off, os.SEEK_SET)
325     s := new(Section)
326     switch f.Class {
327     case ELFCLASS32:
328         sh := new(Section32)
329         if err := binary.Read(sr, f.ByteOrder)
330             return nil, err
331     }
332     names[i] = sh.Name
333     s.SectionHeader = SectionHeader{
334         Type: SectionType(sh.Type),
335         Flags: SectionFlag(sh.Flags),
336         Addr: uint64(sh.Addr),
337         Offset: uint64(sh.Off),
338         Size: uint64(sh.Size),
339         Link: uint32(sh.Link),
340         Info: uint32(sh.Info),

```

```

341             Addralign: uint64(sh.Addrali
342             Entsize:   uint64(sh.Entsize
343         }
344     case ELFCLASS64:
345         sh := new(Section64)
346         if err := binary.Read(sr, f.ByteOrde
347             return nil, err
348     }
349     names[i] = sh.Name
350     s.SectionHeader = SectionHeader{
351         Type:      SectionType(sh.Ty
352         Flags:    SectionFlag(sh.Fl
353         Offset:   uint64(sh.Off),
354         Size:     uint64(sh.Size),
355         Addr:     uint64(sh.Addr),
356         Link:     uint32(sh.Link),
357         Info:     uint32(sh.Info),
358         Addralign: uint64(sh.Addrali
359         Entsize:  uint64(sh.Entsize
360     }
361 }
362 s.sr = io.NewSectionReader(r, int64(s.Offset
363 s.ReaderAt = s.sr
364 f.Sections[i] = s
365 }
366
367 // Load section header string table.
368 shstrtab, err := f.Sections[shstrndx].Data()
369 if err != nil {
370     return nil, err
371 }
372 for i, s := range f.Sections {
373     var ok bool
374     s.Name, ok = getString(shstrtab, int(names[i
375     if !ok {
376         return nil, &FormatError{shoff + int
377     }
378 }
379
380 return f, nil
381 }
382
383 // getSymbols returns a slice of Symbols from parsing the sy
384 // with the given type, along with the associated string tab
385 func (f *File) getSymbols(typ SectionType) ([]Symbol, []byte
386     switch f.Class {
387     case ELFCLASS64:
388         return f.getSymbols64(typ)
389
390     case ELFCLASS32:

```

```

391         return f.getSymbols32(typ)
392     }
393
394     return nil, nil, errors.New("not implemented")
395 }
396
397 func (f *File) getSymbols32(typ SectionType) ([]Symbol, []by
398     symtabSection := f.SectionByType(typ)
399     if symtabSection == nil {
400         return nil, nil, errors.New("no symbol secti
401     }
402
403     data, err := symtabSection.Data()
404     if err != nil {
405         return nil, nil, errors.New("cannot load sym
406     }
407     symtab := bytes.NewBuffer(data)
408     if symtab.Len()%Sym32Size != 0 {
409         return nil, nil, errors.New("length of symbo
410     }
411
412     strdata, err := f.stringTable(symtabSection.Link)
413     if err != nil {
414         return nil, nil, errors.New("cannot load str
415     }
416
417     // The first entry is all zeros.
418     var skip [Sym32Size]byte
419     symtab.Read(skip[0:])
420
421     symbols := make([]Symbol, symtab.Len()/Sym32Size)
422
423     i := 0
424     var sym Sym32
425     for symtab.Len() > 0 {
426         binary.Read(symtab, f.ByteOrder, &sym)
427         str, _ := getString(strdata, int(sym.Name))
428         symbols[i].Name = str
429         symbols[i].Info = sym.Info
430         symbols[i].Other = sym.Other
431         symbols[i].Section = SectionIndex(sym.Shndx)
432         symbols[i].Value = uint64(sym.Value)
433         symbols[i].Size = uint64(sym.Size)
434         i++
435     }
436
437     return symbols, strdata, nil
438 }
439

```

```

440 func (f *File) getSymbols64(typ SectionType) ([]Symbol, []by
441     symtabSection := f.SectionByType(typ)
442     if symtabSection == nil {
443         return nil, nil, errors.New("no symbol secti
444     }
445
446     data, err := symtabSection.Data()
447     if err != nil {
448         return nil, nil, errors.New("cannot load sym
449     }
450     symtab := bytes.NewBuffer(data)
451     if symtab.Len()%Sym64Size != 0 {
452         return nil, nil, errors.New("length of symbo
453     }
454
455     strdata, err := f.stringTable(symtabSection.Link)
456     if err != nil {
457         return nil, nil, errors.New("cannot load str
458     }
459
460     // The first entry is all zeros.
461     var skip [Sym64Size]byte
462     symtab.Read(skip[0:])
463
464     symbols := make([]Symbol, symtab.Len()/Sym64Size)
465
466     i := 0
467     var sym Sym64
468     for symtab.Len() > 0 {
469         binary.Read(symtab, f.ByteOrder, &sym)
470         str, _ := getString(strdata, int(sym.Name))
471         symbols[i].Name = str
472         symbols[i].Info = sym.Info
473         symbols[i].Other = sym.Other
474         symbols[i].Section = SectionIndex(sym.Shndx)
475         symbols[i].Value = sym.Value
476         symbols[i].Size = sym.Size
477         i++
478     }
479
480     return symbols, strdata, nil
481 }
482
483 // getString extracts a string from an ELF string table.
484 func getString(section []byte, start int) (string, bool) {
485     if start < 0 || start >= len(section) {
486         return "", false
487     }
488

```

```

489         for end := start; end < len(section); end++ {
490             if section[end] == 0 {
491                 return string(section[start:end]), t
492             }
493         }
494     return "", false
495 }
496
497 // Section returns a section with the given name, or nil if
498 // section exists.
499 func (f *File) Section(name string) *Section {
500     for _, s := range f.Sections {
501         if s.Name == name {
502             return s
503         }
504     }
505     return nil
506 }
507
508 // applyRelocations applies relocations to dst. rels is a re
509 // in RELA format.
510 func (f *File) applyRelocations(dst []byte, rels []byte) err
511     if f.Class == ELFCLASS64 && f.Machine == EM_X86_64 {
512         return f.applyRelocationsAMD64(dst, rels)
513     }
514
515     return errors.New("not implemented")
516 }
517
518 func (f *File) applyRelocationsAMD64(dst []byte, rels []byte)
519     if len(rels)%Sym64Size != 0 {
520         return errors.New("length of relocation sect
521     }
522
523     symbols, _, err := f.getSymbols(SHT_SYMTAB)
524     if err != nil {
525         return err
526     }
527
528     b := bytes.NewBuffer(rels)
529     var rela Rela64
530
531     for b.Len() > 0 {
532         binary.Read(b, f.ByteOrder, &rela)
533         symNo := rela.Info >> 32
534         t := R_X86_64(rela.Info & 0xffff)
535
536         if symNo >= uint64(len(symbols)) {
537             continue
538         }

```

```

539         sym := &symbols[symNo]
540         if SymType(sym.Info&0xf) != STT_SECTION {
541             // We don't handle non-section reloc
542             continue
543         }
544
545         switch t {
546         case R_X86_64_64:
547             if rela.Off+8 >= uint64(len(dst)) ||
548                 continue
549             }
550             f.ByteOrder.PutUint64(dst[rela.Off:r
551         case R_X86_64_32:
552             if rela.Off+4 >= uint64(len(dst)) ||
553                 continue
554             }
555             f.ByteOrder.PutUint32(dst[rela.Off:r
556         }
557     }
558
559     return nil
560 }
561
562 func (f *File) DWARF() (*dwarf.Data, error) {
563     // There are many other DWARF sections, but these
564     // are the required ones, and the debug/dwarf packag
565     // does not use the others, so don't bother loading
566     var names = [...]string{"abbrev", "info", "str"}
567     var dat [len(names)][[]byte
568     for i, name := range names {
569         name = ".debug_" + name
570         s := f.Section(name)
571         if s == nil {
572             continue
573         }
574         b, err := s.Data()
575         if err != nil && uint64(len(b)) < s.Size {
576             return nil, err
577         }
578         dat[i] = b
579     }
580
581     // If there's a relocation table for .debug_info, we
582     // now otherwise the data in .debug_info is invalid
583     rela := f.Section(".rela.debug_info")
584     if rela != nil && rela.Type == SHT_RELA && f.Machine
585         data, err := rela.Data()
586         if err != nil {
587             return nil, err

```

```

588     }
589     err = f.applyRelocations(dat[1], data)
590     if err != nil {
591         return nil, err
592     }
593 }
594
595     abbrev, info, str := dat[0], dat[1], dat[2]
596     return dwarf.New(abbrev, nil, nil, info, nil, nil, n
597 }
598
599 // Symbols returns the symbol table for f.
600 func (f *File) Symbols() ([]Symbol, error) {
601     sym, _, err := f.getSymbols(SHT_SYMTAB)
602     return sym, err
603 }
604
605 type ImportedSymbol struct {
606     Name     string
607     Version string
608     Library string
609 }
610
611 // ImportedSymbols returns the names of all symbols
612 // referred to by the binary f that are expected to be
613 // satisfied by other libraries at dynamic load time.
614 // It does not return weak symbols.
615 func (f *File) ImportedSymbols() ([]ImportedSymbol, error) {
616     sym, str, err := f.getSymbols(SHT_DYNSYM)
617     if err != nil {
618         return nil, err
619     }
620     f.gnuVersionInit(str)
621     var all []ImportedSymbol
622     for i, s := range sym {
623         if ST_BIND(s.Info) == STB_GLOBAL && s.Sectio
624             all = append(all, ImportedSymbol{Nar
625                 f.gnuVersion(i, &all[len(all)-1])
626         }
627     }
628     return all, nil
629 }
630
631 type verneed struct {
632     File string
633     Name string
634 }
635
636 // gnuVersionInit parses the GNU version tables

```

```

637 // for use by calls to gnuVersion.
638 func (f *File) gnuVersionInit(str []byte) {
639     // Accumulate verneed information.
640     vn := f.SectionByType(SHT_GNU_VERNEED)
641     if vn == nil {
642         return
643     }
644     d, _ := vn.Data()
645
646     var need []verneed
647     i := 0
648     for {
649         if i+16 > len(d) {
650             break
651         }
652         vers := f.ByteOrder.Uint16(d[i : i+2])
653         if vers != 1 {
654             break
655         }
656         cnt := f.ByteOrder.Uint16(d[i+2 : i+4])
657         fileoff := f.ByteOrder.Uint32(d[i+4 : i+8])
658         aux := f.ByteOrder.Uint32(d[i+8 : i+12])
659         next := f.ByteOrder.Uint32(d[i+12 : i+16])
660         file, _ := getString(str, int(fileoff))
661
662         var name string
663         j := i + int(aux)
664         for c := 0; c < int(cnt); c++ {
665             if j+16 > len(d) {
666                 break
667             }
668             // hash := f.ByteOrder.Uint32(d[j:j+
669             // flags := f.ByteOrder.Uint16(d[j+4
670             other := f.ByteOrder.Uint16(d[j+6 :
671             nameoff := f.ByteOrder.Uint32(d[j+8
672             next := f.ByteOrder.Uint32(d[j+12 :
673             name, _ = getString(str, int(nameoff
674             ndx := int(other)
675             if ndx >= len(need) {
676                 a := make([]verneed, 2*(ndx+
677                 copy(a, need)
678                 need = a
679             }
680
681             need[ndx] = verneed{file, name}
682             if next == 0 {
683                 break
684             }
685             j += int(next)
686         }

```

```

687
688         if next == 0 {
689             break
690         }
691         i += int(next)
692     }
693
694     // Versym parallels symbol table, indexing into vern
695     vs := f.SectionByType(SHT_GNU_VERSYM)
696     if vs == nil {
697         return
698     }
699     d, _ = vs.Data()
700
701     f.gnuNeed = need
702     f.gnuVersym = d
703 }
704
705 // gnuVersion adds Library and Version information to sym,
706 // which came from offset i of the symbol table.
707 func (f *File) gnuVersion(i int, sym *ImportedSymbol) {
708     // Each entry is two bytes; skip undef entry at begi
709     i = (i + 1) * 2
710     if i >= len(f.gnuVersym) {
711         return
712     }
713     j := int(f.ByteOrder.Uint16(f.gnuVersym[i:]))
714     if j < 2 || j >= len(f.gnuNeed) {
715         return
716     }
717     n := &f.gnuNeed[j]
718     sym.Library = n.File
719     sym.Version = n.Name
720 }
721
722 // ImportedLibraries returns the names of all libraries
723 // referred to by the binary f that are expected to be
724 // linked with the binary at dynamic link time.
725 func (f *File) ImportedLibraries() ([]string, error) {
726     ds := f.SectionByType(SHT_DYNAMIC)
727     if ds == nil {
728         // not dynamic, so no libraries
729         return nil, nil
730     }
731     d, err := ds.Data()
732     if err != nil {
733         return nil, err
734     }
735     str, err := f.stringTable(ds.Link)

```

```

736     if err != nil {
737         return nil, err
738     }
739     var all []string
740     for len(d) > 0 {
741         var tag DynTag
742         var value uint64
743         switch f.Class {
744             case ELFCLASS32:
745                 tag = DynTag(f.ByteOrder.Uint32(d[0:
746                 value = uint64(f.ByteOrder.Uint32(d[
747                 d = d[8:]
748             case ELFCLASS64:
749                 tag = DynTag(f.ByteOrder.Uint64(d[0:
750                 value = f.ByteOrder.Uint64(d[8:16])
751                 d = d[16:]
752             }
753             if tag == DT_NEEDED {
754                 s, ok := getString(str, int(value))
755                 if ok {
756                     all = append(all, s)
757                 }
758             }
759         }
760     }
761     return all, nil
762 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/gosym/pclntab.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6  * Line tables
7  */
8
9 package gosym
10
11 import "encoding/binary"
12
13 type LineTable struct {
14     Data []byte
15     PC    uint64
16     Line int
17 }
18
19 // TODO(rsc): Need to pull in quantum from architecture defi
20 const quantum = 1
21
22 func (t *LineTable) parse(targetPC uint64, targetLine int) (
23     // The PC/line table can be thought of as a sequence
24     // <pc update>* <line update>
25     // batches. Each update batch results in a (pc, lin
26     // where line applies to every PC from pc up to but
27     // including the pc of the next pair.
28     //
29     // Here we process each update individually, which s
30     // the code, but makes the corner cases more confusi
31     b, pc, line = t.Data, t.PC, t.Line
32     for pc <= targetPC && line != targetLine && len(b) >
33         code := b[0]
34         b = b[1:]
35         switch {
36         case code == 0:
37             if len(b) < 4 {
38                 b = b[0:0]
39                 break
40             }
41             val := binary.BigEndian.Uint32(b)
```

```

42             b = b[4:]
43             line += int(val)
44         case code <= 64:
45             line += int(code)
46         case code <= 128:
47             line -= int(code - 64)
48         default:
49             pc += quantum * uint64(code-128)
50             continue
51     }
52     pc += quantum
53 }
54 return b, pc, line
55 }
56
57 func (t *LineTable) slice(pc uint64) *LineTable {
58     data, pc, line := t.parse(pc, -1)
59     return &LineTable{data, pc, line}
60 }
61
62 func (t *LineTable) PCToLine(pc uint64) int {
63     _, _, line := t.parse(pc, -1)
64     return line
65 }
66
67 func (t *LineTable) LineToPC(line int, maxpc uint64) uint64
68     _, pc, line1 := t.parse(maxpc, line)
69     if line1 != line {
70         return 0
71     }
72     // Subtract quantum from PC to account for post-line
73     return pc - quantum
74 }
75
76 // NewLineTable returns a new PC/line table
77 // corresponding to the encoded data.
78 // Text must be the start address of the
79 // corresponding text segment.
80 func NewLineTable(data []byte, text uint64) *LineTable {
81     return &LineTable{data, text, 0}
82 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/gosym/symtab.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package gosym implements access to the Go symbol
6 // and line number tables embedded in Go binaries generated
7 // by the gc compilers.
8 package gosym
9
10 // The table format is a variant of the format used in Plan
11 // format, documented at http://plan9.bell-labs.com/magic/ma
12 // The best reference for the differences between the Plan 9
13 // and the Go format is the runtime source, specifically ../
14
15 import (
16     "encoding/binary"
17     "fmt"
18     "strconv"
19     "strings"
20 )
21
22 /*
23  * Symbols
24  */
25
26 // A Sym represents a single symbol table entry.
27 type Sym struct {
28     Value  uint64
29     Type   byte
30     Name   string
31     GoType uint64
32     // If this symbol is a function symbol, the correspo
33     Func  *Func
34 }
35
36 // Static returns whether this symbol is static (not visible
37 func (s *Sym) Static() bool { return s.Type >= 'a' }
38
39 // PackageName returns the package part of the symbol name,
40 // or the empty string if there is none.
41 func (s *Sym) PackageName() string {
```

```

42         if i := strings.Index(s.Name, "."); i != -1 {
43             return s.Name[0:i]
44         }
45         return ""
46     }
47
48     // ReceiverName returns the receiver type name of this symbol
49     // or the empty string if there is none.
50     func (s *Sym) ReceiverName() string {
51         l := strings.Index(s.Name, ".")
52         r := strings.LastIndex(s.Name, ".")
53         if l == -1 || r == -1 || l == r {
54             return ""
55         }
56         return s.Name[l+1 : r]
57     }
58
59     // BaseName returns the symbol name without the package or r
60     func (s *Sym) BaseName() string {
61         if i := strings.LastIndex(s.Name, "."); i != -1 {
62             return s.Name[i+1:]
63         }
64         return s.Name
65     }
66
67     // A Func collects information about a single function.
68     type Func struct {
69         Entry uint64
70         *Sym
71         End      uint64
72         Params   []*Sym
73         Locals   []*Sym
74         FrameSize int
75         LineTable *LineTable
76         Obj      *Obj
77     }
78
79     // An Obj represents a single object file.
80     type Obj struct {
81         Funcs []Func
82         Paths []Sym
83     }
84
85     /*
86     * Symbol tables
87     */
88
89     // Table represents a Go symbol table. It stores all of the
90     // symbols decoded from the program and provides methods to
91     // between symbols, names, and addresses.

```

```

92 type Table struct {
93     Syms []Sym
94     Funcs []Func
95     Files map[string]*Obj
96     Objs []Obj
97     //     textEnd uint64;
98 }
99
100 type sym struct {
101     value uint32
102     gotype uint32
103     typ byte
104     name []byte
105 }
106
107 func walksymtab(data []byte, fn func(sym) error) error {
108     var s sym
109     p := data
110     for len(p) >= 6 {
111         s.value = binary.BigEndian.Uint32(p[0:4])
112         typ := p[4]
113         if typ&0x80 == 0 {
114             return &DecodingError{len(data) - le
115         }
116         typ &^= 0x80
117         s.typ = typ
118         p = p[5:]
119         var i int
120         var nnul int
121         for i = 0; i < len(p); i++ {
122             if p[i] == 0 {
123                 nnul = 1
124                 break
125             }
126         }
127         switch typ {
128         case 'z', 'Z':
129             p = p[i+nnul:]
130             for i = 0; i+2 <= len(p); i += 2 {
131                 if p[i] == 0 && p[i+1] == 0
132                     nnul = 2
133                     break
134             }
135         }
136     }
137     if i+nnul+4 > len(p) {
138         return &DecodingError{len(data), "un
139     }
140     s.name = p[0:i]

```

```

141             i += nnul
142             s.gotype = binary.BigEndian.Uint32(p[i : i+4]
143             p = p[i+4:]
144             fn(s)
145         }
146         return nil
147     }
148
149     // NewTable decodes the Go symbol table in data,
150     // returning an in-memory representation.
151     func NewTable(symtab []byte, pcln *LineTable) (*Table, error)
152         var n int
153         err := walksymtab(symtab, func(s sym) error {
154             n++
155             return nil
156         })
157         if err != nil {
158             return nil, err
159         }
160
161         var t Table
162         fname := make(map[uint16]string)
163         t.Syms = make([]Sym, 0, n)
164         nf := 0
165         nz := 0
166         lasttyp := uint8(0)
167         err = walksymtab(symtab, func(s sym) error {
168             n := len(t.Syms)
169             t.Syms = t.Syms[0 : n+1]
170             ts := &t.Syms[n]
171             ts.Type = s.typ
172             ts.Value = uint64(s.value)
173             ts.GoType = uint64(s.gotype)
174             switch s.typ {
175             default:
176                 // rewrite name to use . instead of
177                 w := 0
178                 b := s.name
179                 for i := 0; i < len(b); i++ {
180                     if b[i] == 0xc2 && i+1 < len
181                         i++
182                         b[i] = '.'
183                 }
184                 b[w] = b[i]
185                 w++
186             }
187             ts.Name = string(s.name[0:w])
188         case 'z', 'Z':
189             if lasttyp != 'z' && lasttyp != 'Z'

```

```

190                                     nz++
191                                 }
192                                 for i := 0; i < len(s.name); i += 2
193                                     eltIdx := binary.BigEndian.U
194                                     elt, ok := fname[eltIdx]
195                                     if !ok {
196                                         return &DecodingError
197                                     }
198                                     if n := len(ts.Name); n > 0
199                                         ts.Name += "/"
200                                 }
201                                 ts.Name += elt
202                             }
203                         }
204                         switch s.typ {
205                         case 'T', 't', 'L', 'l':
206                             nf++
207                         case 'f':
208                             fname[uint16(s.value)] = ts.Name
209                         }
210                         lasttyp = s.typ
211                         return nil
212                 })
213                 if err != nil {
214                     return nil, err
215                 }
216
217                 t.Funcs = make([]Func, 0, nf)
218                 t.Objjs = make([]Obj, 0, nz)
219                 t.Files = make(map[string]*Obj)
220
221                 // Count text symbols and attach frame sizes, parame
222                 // locals to them. Also, find object file boundarie
223                 var obj *Obj
224                 lastf := 0
225                 for i := 0; i < len(t.Syms); i++ {
226                     sym := &t.Syms[i]
227                     switch sym.Type {
228                     case 'Z', 'z': // path symbol
229                         // Finish the current object
230                         if obj != nil {
231                             obj.Funcs = t.Funcs[lastf:]
232                         }
233                         lastf = len(t.Funcs)
234
235                         // Start new object
236                         n := len(t.Objjs)
237                         t.Objjs = t.Objjs[0 : n+1]
238                         obj = &t.Objjs[n]
239

```

```

240         // Count & copy path symbols
241         var end int
242         for end = i + 1; end < len(t.Syms);
243             if c := t.Syms[end].Type; c
244                 break
245             }
246     }
247     obj.Paths = t.Syms[i:end]
248     i = end - 1 // loop will i++
249
250     // Record file names
251     depth := 0
252     for j := range obj.Paths {
253         s := &obj.Paths[j]
254         if s.Name == "" {
255             depth--
256         } else {
257             if depth == 0 {
258                 t.Files[s.Na
259             }
260             depth++
261         }
262     }
263
264     case 'T', 't', 'L', 'l': // text symbol
265         if n := len(t.Funcs); n > 0 {
266             t.Funcs[n-1].End = sym.Value
267         }
268         if sym.Name == "etext" {
269             continue
270         }
271
272     // Count parameter and local (auto)
273     var np, na int
274     var end int
275     countloop:
276         for end = i + 1; end < len(t.Syms);
277             switch t.Syms[end].Type {
278                 case 'T', 't', 'L', 'l', 'Z'
279                     break countloop
280                 case 'p':
281                     np++
282                 case 'a':
283                     na++
284             }
285         }
286
287     // Fill in the function symbol
288     n := len(t.Funcs)

```

```

289         t.Funcs = t.Funcs[0 : n+1]
290         fn := &t.Funcs[n]
291         sym.Func = fn
292         fn.Params = make([]*Sym, 0, np)
293         fn.Locals = make([]*Sym, 0, na)
294         fn.Sym = sym
295         fn.Entry = sym.Value
296         fn.Obj = obj
297         if pcln != nil {
298             fn.LineTable = pcln.slice(fn
299                 pcln = fn.LineTable
300         }
301         for j := i; j < end; j++ {
302             s := &t.Syms[j]
303             switch s.Type {
304             case 'm':
305                 fn.FrameSize = int(s
306             case 'p':
307                 n := len(fn.Params)
308                 fn.Params = fn.Param
309                 fn.Params[n] = s
310             case 'a':
311                 n := len(fn.Locals)
312                 fn.Locals = fn.Local
313                 fn.Locals[n] = s
314             }
315         }
316         i = end - 1 // loop will i++
317     }
318 }
319 if obj != nil {
320     obj.Funcs = t.Funcs[lastf:]
321 }
322 return &t, nil
323 }
324
325 // PCToFunc returns the function containing the program coun
326 // or nil if there is no such function.
327 func (t *Table) PCToFunc(pc uint64) *Func {
328     funcs := t.Funcs
329     for len(funcs) > 0 {
330         m := len(funcs) / 2
331         fn := &funcs[m]
332         switch {
333         case pc < fn.Entry:
334             funcs = funcs[0:m]
335         case fn.Entry <= pc && pc < fn.End:
336             return fn
337         default:

```



```

388     }
389     return nil
390 }
391
392 // LookupFunc returns the text, data, or bss symbol with the
393 // or nil if no such symbol is found.
394 func (t *Table) LookupFunc(name string) *Func {
395     for i := range t.Funcs {
396         f := &t.Funcs[i]
397         if f.Sym.Name == name {
398             return f
399         }
400     }
401     return nil
402 }
403
404 // SymByAddr returns the text, data, or bss symbol starting
405 // TODO(rsc): Allow lookup by any address within the symbol.
406 func (t *Table) SymByAddr(addr uint64) *Sym {
407     // TODO(austin) Maybe make a map
408     for i := range t.Syms {
409         s := &t.Syms[i]
410         switch s.Type {
411             case 'T', 't', 'L', 'l', 'D', 'd', 'B', 'b':
412                 if s.Value == addr {
413                     return s
414                 }
415             }
416     }
417     return nil
418 }
419
420 /*
421  * Object files
422  */
423
424 func (o *Obj) lineFromAline(aline int) (string, int) {
425     type stackEnt struct {
426         path    string
427         start   int
428         offset  int
429         prev   *stackEnt
430     }
431
432     noPath := &stackEnt{"", 0, 0, nil}
433     tos := noPath
434
435     // TODO(austin) I have no idea how 'Z' symbols work,
436     // that they pop the stack.

```

```

437 pathloop:
438     for _, s := range o.Paths {
439         val := int(s.Value)
440         switch {
441             case val > aline:
442                 break pathloop
443
444             case val == 1:
445                 // Start a new stack
446                 tos = &stackEnt{s.Name, val, 0, noPa
447
448             case s.Name == "":
449                 // Pop
450                 if tos == noPath {
451                     return "<malformed symbol ta
452                 }
453                 tos.prev.offset += val - tos.start
454                 tos = tos.prev
455
456             default:
457                 // Push
458                 tos = &stackEnt{s.Name, val, 0, tos}
459         }
460     }
461
462     if tos == noPath {
463         return "", 0
464     }
465     return tos.path, aline - tos.start - tos.offset + 1
466 }
467
468 func (o *Obj) alineFromLine(path string, line int) (int, err
469     if line < 1 {
470         return 0, &UnknownLineError{path, line}
471     }
472
473     for i, s := range o.Paths {
474         // Find this path
475         if s.Name != path {
476             continue
477         }
478
479         // Find this line at this stack level
480         depth := 0
481         var incstart int
482         line += int(s.Value)
483     pathloop:
484         for _, s := range o.Paths[i:] {
485             val := int(s.Value)

```

```

486         switch {
487         case depth == 1 && val >= line:
488             return line - 1, nil
489
490         case s.Name == "":
491             depth--
492             if depth == 0 {
493                 break pathloop
494             } else if depth == 1 {
495                 line += val - incsta
496             }
497
498         default:
499             if depth == 1 {
500                 incstart = val
501             }
502             depth++
503         }
504     }
505     return 0, &UnknownLineError{path, line}
506 }
507 return 0, UnknownFileError(path)
508 }
509
510 /*
511  * Errors
512  */
513
514 // UnknownFileError represents a failure to find the specifi
515 // the symbol table.
516 type UnknownFileError string
517
518 func (e UnknownFileError) Error() string { return "unknown f
519
520 // UnknownLineError represents a failure to map a line to a
521 // counter, either because the line is beyond the bounds of
522 // or because there is no code on the given line.
523 type UnknownLineError struct {
524     File string
525     Line int
526 }
527
528 func (e *UnknownLineError) Error() string {
529     return "no code at " + e.File + ":" + strconv.Itoa(e
530 }
531
532 // DecodingError represents an error during the decoding of
533 // the symbol table.
534 type DecodingError struct {
535     off int

```

```
536         msg string
537         val interface{}
538     }
539
540     func (e *DecodingError) Error() string {
541         msg := e.msg
542         if e.val != nil {
543             msg += fmt.Sprintf(" '%v'", e.val)
544         }
545         msg += fmt.Sprintf(" at byte %#x", e.off)
546         return msg
547     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/macho/file.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package macho implements access to Mach-O object files.
6 package macho
7
8 // High level access to low level data structures.
9
10 import (
11     "bytes"
12     "debug/dwarf"
13     "encoding/binary"
14     "errors"
15     "fmt"
16     "io"
17     "os"
18 )
19
20 // A File represents an open Mach-O file.
21 type File struct {
22     FileHeader
23     ByteOrder binary.ByteOrder
24     Loads      []Load
25     Sections   []*Section
26
27     Symtab *Symtab
28     Dysymtab *Dysymtab
29
30     closer io.Closer
31 }
32
33 // A Load represents any Mach-O load command.
34 type Load interface {
35     Raw() []byte
36 }
37
38 // A LoadBytes is the uninterpreted bytes of a Mach-O load c
39 type LoadBytes []byte
40
41 func (b LoadBytes) Raw() []byte { return b }
```

```

42
43 // A SegmentHeader is the header for a Mach-0 32-bit or 64-b
44 type SegmentHeader struct {
45     Cmd      LoadCmd
46     Len      uint32
47     Name     string
48     Addr     uint64
49     Memsz    uint64
50     Offset   uint64
51     Filesz   uint64
52     Maxprot  uint32
53     Prot     uint32
54     Nsect    uint32
55     Flag     uint32
56 }
57
58 // A Segment represents a Mach-0 32-bit or 64-bit load segme
59 type Segment struct {
60     LoadBytes
61     SegmentHeader
62
63     // Embed ReaderAt for ReadAt method.
64     // Do not embed SectionReader directly
65     // to avoid having Read and Seek.
66     // If a client wants Read and Seek it must use
67     // Open() to avoid fighting over the seek offset
68     // with other clients.
69     io.ReaderAt
70     sr *io.SectionReader
71 }
72
73 // Data reads and returns the contents of the segment.
74 func (s *Segment) Data() ([]byte, error) {
75     dat := make([]byte, s.sr.Size())
76     n, err := s.sr.ReadAt(dat, 0)
77     return dat[0:n], err
78 }
79
80 // Open returns a new ReadSeeker reading the segment.
81 func (s *Segment) Open() io.ReadSeeker { return io.NewSectio
82
83 type SectionHeader struct {
84     Name     string
85     Seg      string
86     Addr     uint64
87     Size     uint64
88     Offset   uint32
89     Align    uint32
90     Reloff   uint32
91     Nreloc   uint32

```

```

92         Flags    uint32
93     }
94
95     type Section struct {
96         SectionHeader
97
98         // Embed ReaderAt for ReadAt method.
99         // Do not embed SectionReader directly
100        // to avoid having Read and Seek.
101        // If a client wants Read and Seek it must use
102        // Open() to avoid fighting over the seek offset
103        // with other clients.
104        io.ReaderAt
105        sr *io.SectionReader
106    }
107
108    // Data reads and returns the contents of the Mach-0 section
109    func (s *Section) Data() ([]byte, error) {
110        dat := make([]byte, s.sr.Size())
111        n, err := s.sr.ReadAt(dat, 0)
112        return dat[0:n], err
113    }
114
115    // Open returns a new ReadSeeker reading the Mach-0 section.
116    func (s *Section) Open() io.ReadSeeker { return io.NewSectionReader(s.sr, s.offset) }
117
118    // A Dylib represents a Mach-0 load dynamic library command.
119    type Dylib struct {
120        LoadBytes
121        Name          string
122        Time           uint32
123        CurrentVersion uint32
124        CompatVersion  uint32
125    }
126
127    // A Symtab represents a Mach-0 symbol table command.
128    type Symtab struct {
129        LoadBytes
130        SymtabCmd
131        Syms []Symbol
132    }
133
134    // A Dysymtab represents a Mach-0 dynamic symbol table command.
135    type Dysymtab struct {
136        LoadBytes
137        DysymtabCmd
138        IndirectSyms []uint32 // indices into Symtab.Syms
139    }
140

```

```

141  /*
142   * Mach-0 reader
143   */
144
145  type FormatError struct {
146      off int64
147      msg string
148      val interface{}
149  }
150
151  func (e *FormatError) Error() string {
152      msg := e.msg
153      if e.val != nil {
154          msg += fmt.Sprintf(" '%v'", e.val)
155      }
156      msg += fmt.Sprintf(" in record at byte %#x", e.off)
157      return msg
158  }
159
160  // Open opens the named file using os.Open and prepares it f
161  func Open(name string) (*File, error) {
162      f, err := os.Open(name)
163      if err != nil {
164          return nil, err
165      }
166      ff, err := NewFile(f)
167      if err != nil {
168          f.Close()
169          return nil, err
170      }
171      ff.closer = f
172      return ff, nil
173  }
174
175  // Close closes the File.
176  // If the File was created using NewFile directly instead of
177  // Close has no effect.
178  func (f *File) Close() error {
179      var err error
180      if f.closer != nil {
181          err = f.closer.Close()
182          f.closer = nil
183      }
184      return err
185  }
186
187  // NewFile creates a new File for accessing a Mach-0 binary
188  // The Mach-0 binary is expected to start at position 0 in t
189  func NewFile(r io.ReaderAt) (*File, error) {

```

```

190     f := new(File)
191     sr := io.NewSectionReader(r, 0, 1<<63-1)
192
193     // Read and decode Mach magic to determine byte orde
194     // Magic32 and Magic64 differ only in the bottom bit
195     var ident [4]byte
196     if _, err := r.ReadAt(ident[0:], 0); err != nil {
197         return nil, err
198     }
199     be := binary.BigEndian.Uint32(ident[0:])
200     le := binary.LittleEndian.Uint32(ident[0:])
201     switch Magic32 &^ 1 {
202     case be &^ 1:
203         f.ByteOrder = binary.BigEndian
204         f.Magic = be
205     case le &^ 1:
206         f.ByteOrder = binary.LittleEndian
207         f.Magic = le
208     default:
209         return nil, &FormatError{0, "invalid magic n
210     }
211
212     // Read entire file header.
213     if err := binary.Read(sr, f.ByteOrder, &f.FileHeader
214         return nil, err
215     }
216
217     // Then load commands.
218     offset := int64(fileHeaderSize32)
219     if f.Magic == Magic64 {
220         offset = fileHeaderSize64
221     }
222     dat := make([]byte, f.Cmdsz)
223     if _, err := r.ReadAt(dat, offset); err != nil {
224         return nil, err
225     }
226     f.Loads = make([]Load, f.Ncmd)
227     bo := f.ByteOrder
228     for i := range f.Loads {
229         // Each load command begins with uint32 comm
230         if len(dat) < 8 {
231             return nil, &FormatError{offset, "co
232         }
233         cmd, siz := LoadCmd(bo.Uint32(dat[0:4])), bo
234         if siz < 8 || siz > uint32(len(dat)) {
235             return nil, &FormatError{offset, "in
236         }
237         var cmddat []byte
238         cmddat, dat = dat[0:siz], dat[siz:]
239         offset += int64(siz)

```

```

240     var s *Segment
241     switch cmd {
242     default:
243         f.Loads[i] = LoadBytes(cmddat)
244
245     case LoadCmdDylib:
246         var hdr DylibCmd
247         b := bytes.NewBuffer(cmddat)
248         if err := binary.Read(b, bo, &hdr);
249             return nil, err
250         }
251         l := new(Dylib)
252         if hdr.Name >= uint32(len(cmddat)) {
253             return nil, &FormatError{off
254         }
255         l.Name = cstring(cmddat[hdr.Name:])
256         l.Time = hdr.Time
257         l.CurrentVersion = hdr.CurrentVersio
258         l.CompatVersion = hdr.CompatVersion
259         l.LoadBytes = LoadBytes(cmddat)
260         f.Loads[i] = l
261
262     case LoadCmdSymtab:
263         var hdr SymtabCmd
264         b := bytes.NewBuffer(cmddat)
265         if err := binary.Read(b, bo, &hdr);
266             return nil, err
267         }
268         strtab := make([]byte, hdr.Strsize)
269         if _, err := r.ReadAt(strtab, int64(
270             return nil, err
271         }
272         var symsz int
273         if f.Magic == Magic64 {
274             symsz = 16
275         } else {
276             symsz = 12
277         }
278         symdat := make([]byte, int(hdr.Nsyms
279         if _, err := r.ReadAt(symdat, int64(
280             return nil, err
281         }
282         st, err := f.parseSymtab(symdat, str
283         if err != nil {
284             return nil, err
285         }
286         f.Loads[i] = st
287         f.Symtab = st
288

```

```

289         case LoadCmdDysymtab:
290             var hdr DysymtabCmd
291             b := bytes.NewBuffer(cmddat)
292             if err := binary.Read(b, bo, &hdr);
293                 return nil, err
294             }
295             dat := make([]byte, hdr.Nindirectsym
296             if _, err := r.ReadAt(dat, int64(hdr
297                 return nil, err
298             }
299             x := make([]uint32, hdr.Nindirectsym
300             if err := binary.Read(bytes.NewBuffe
301                 return nil, err
302             }
303             st := new(Dysymtab)
304             st.LoadBytes = LoadBytes(cmddat)
305             st.DysymtabCmd = hdr
306             st.IndirectSyms = x
307             f.Loads[i] = st
308             f.Dysymtab = st
309
310         case LoadCmdSegment:
311             var seg32 Segment32
312             b := bytes.NewBuffer(cmddat)
313             if err := binary.Read(b, bo, &seg32)
314                 return nil, err
315             }
316             s = new(Segment)
317             s.LoadBytes = cmddat
318             s.Cmd = cmd
319             s.Len = siz
320             s.Name = cstring(seg32.Name[0:])
321             s.Addr = uint64(seg32.Addr)
322             s.Memsz = uint64(seg32.Memsz)
323             s.Offset = uint64(seg32.Offset)
324             s.Filesz = uint64(seg32.Filesz)
325             s.Maxprot = seg32.Maxprot
326             s.Prot = seg32.Prot
327             s.Nsect = seg32.Nsect
328             s.Flag = seg32.Flag
329             f.Loads[i] = s
330             for i := 0; i < int(s.Nsect); i++ {
331                 var sh32 Section32
332                 if err := binary.Read(b, bo,
333                     return nil, err
334                 }
335                 sh := new(Section)
336                 sh.Name = cstring(sh32.Name[
337                 sh.Seg = cstring(sh32.Seg[0:

```

```

338         sh.Addr = uint64(sh32.Addr)
339         sh.Size = uint64(sh32.Size)
340         sh.Offset = sh32.Offset
341         sh.Align = sh32.Align
342         sh.Reloff = sh32.Reloff
343         sh.Nreloc = sh32.Nreloc
344         sh.Flags = sh32.Flags
345         f.pushSection(sh, r)
346     }
347
348     case LoadCmdSegment64:
349         var seg64 Segment64
350         b := bytes.NewBuffer(cmddat)
351         if err := binary.Read(b, bo, &seg64)
352             return nil, err
353     }
354     s = new(Segment)
355     s.LoadBytes = cmddat
356     s.Cmd = cmd
357     s.Len = siz
358     s.Name = cstring(seg64.Name[0:])
359     s.Addr = seg64.Addr
360     s.Memsz = seg64.Memsz
361     s.Offset = seg64.Offset
362     s.Filesz = seg64.Filesz
363     s.Maxprot = seg64.Maxprot
364     s.Prot = seg64.Prot
365     s.Nsect = seg64.Nsect
366     s.Flag = seg64.Flag
367     f.Loads[i] = s
368     for i := 0; i < int(s.Nsect); i++ {
369         var sh64 Section64
370         if err := binary.Read(b, bo,
371             return nil, err
372     }
373     sh := new(Section)
374     sh.Name = cstring(sh64.Name[
375     sh.Seg = cstring(sh64.Seg[0:
376     sh.Addr = sh64.Addr
377     sh.Size = sh64.Size
378     sh.Offset = sh64.Offset
379     sh.Align = sh64.Align
380     sh.Reloff = sh64.Reloff
381     sh.Nreloc = sh64.Nreloc
382     sh.Flags = sh64.Flags
383     f.pushSection(sh, r)
384     }
385 }
386 if s != nil {
387     s.sr = io.NewSectionReader(r, int64(

```

```

388             s.ReaderAt = s.sr
389         }
390     }
391     return f, nil
392 }
393
394 func (f *File) parseSymtab(symdat, strtab, cmddat []byte, hd
395     bo := f.ByteOrder
396     symtab := make([]Symbol, hdr.Nsyms)
397     b := bytes.NewBuffer(symdat)
398     for i := range symtab {
399         var n Nlist64
400         if f.Magic == Magic64 {
401             if err := binary.Read(b, bo, &n); er
402                 return nil, err
403         }
404         } else {
405             var n32 Nlist32
406             if err := binary.Read(b, bo, &n32);
407                 return nil, err
408         }
409         n.Name = n32.Name
410         n.Type = n32.Type
411         n.Sect = n32.Sect
412         n.Desc = n32.Desc
413         n.Value = uint64(n32.Value)
414     }
415     sym := &symtab[i]
416     if n.Name >= uint32(len(strtab)) {
417         return nil, &FormatError{offset, "in
418     }
419     sym.Name = cstring(strtab[n.Name:])
420     sym.Type = n.Type
421     sym.Sect = n.Sect
422     sym.Desc = n.Desc
423     sym.Value = n.Value
424 }
425 st := new(Symtab)
426 st.LoadBytes = LoadBytes(cmddat)
427 st.Syms = symtab
428 return st, nil
429 }
430
431 func (f *File) pushSection(sh *Section, r io.ReaderAt) {
432     f.Sections = append(f.Sections, sh)
433     sh.sr = io.NewSectionReader(r, int64(sh.Offset), int
434     sh.ReaderAt = sh.sr
435 }
436

```

```

437 func cstring(b []byte) string {
438     var i int
439     for i = 0; i < len(b) && b[i] != 0; i++ {
440     }
441     return string(b[0:i])
442 }
443
444 // Segment returns the first Segment with the given name, or
445 func (f *File) Segment(name string) *Segment {
446     for _, l := range f.Loads {
447         if s, ok := l.(*Segment); ok && s.Name == name {
448             return s
449         }
450     }
451     return nil
452 }
453
454 // Section returns the first section with the given name, or
455 // section exists.
456 func (f *File) Section(name string) *Section {
457     for _, s := range f.Sections {
458         if s.Name == name {
459             return s
460         }
461     }
462     return nil
463 }
464
465 // DWARF returns the DWARF debug information for the Mach-O
466 func (f *File) DWARF() (*dwarf.Data, error) {
467     // There are many other DWARF sections, but these
468     // are the required ones, and the debug/dwarf package
469     // does not use the others, so don't bother loading
470     var names = [...]string{"abbrev", "info", "str"}
471     var dat [len(names)][[]byte
472     for i, name := range names {
473         name = "__debug_" + name
474         s := f.Section(name)
475         if s == nil {
476             return nil, errors.New("missing Mach
477         }
478         b, err := s.Data()
479         if err != nil && uint64(len(b)) < s.Size {
480             return nil, err
481         }
482         dat[i] = b
483     }
484
485     abbrev, info, str := dat[0], dat[1], dat[2]

```

```

486         return dwarf.New(abbrev, nil, nil, info, nil, nil, n
487     }
488
489     // ImportedSymbols returns the names of all symbols
490     // referred to by the binary f that are expected to be
491     // satisfied by other libraries at dynamic load time.
492     func (f *File) ImportedSymbols() ([]string, error) {
493         if f.Dysymtab == nil || f.Symtab == nil {
494             return nil, &FormatError{0, "missing symbol
495         }
496
497         st := f.Symtab
498         dt := f.Dysymtab
499         var all []string
500         for _, s := range st.Syms[dt.Iundefsym : dt.Iundefsy
501             all = append(all, s.Name)
502         }
503         return all, nil
504     }
505
506     // ImportedLibraries returns the paths of all libraries
507     // referred to by the binary f that are expected to be
508     // linked with the binary at dynamic link time.
509     func (f *File) ImportedLibraries() ([]string, error) {
510         var all []string
511         for _, l := range f.Loads {
512             if lib, ok := l.(*Dylib); ok {
513                 all = append(all, lib.Name)
514             }
515         }
516         return all, nil
517     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/debug/macho/macho.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Mach-0 header data structures
6 // http://developer.apple.com/mac/library/documentation/Deve
7
8 package macho
9
10 import "strconv"
11
12 // A FileHeader represents a Mach-0 file header.
13 type FileHeader struct {
14     Magic    uint32
15     Cpu      Cpu
16     SubCpu   uint32
17     Type     Type
18     Ncmd     uint32
19     Cmdsz    uint32
20     Flags    uint32
21 }
22
23 const (
24     fileHeaderSize32 = 7 * 4
25     fileHeaderSize64 = 8 * 4
26 )
27
28 const (
29     Magic32 uint32 = 0xfeedface
30     Magic64 uint32 = 0xfeedfacf
31 )
32
33 // A Type is a Mach-0 file type, either an object or an exec
34 type Type uint32
35
36 const (
37     TypeObj   Type = 1
38     TypeExec Type = 2
39 )
40
41 // A Cpu is a Mach-0 cpu type.
```

```

42 type Cpu uint32
43
44 const (
45     Cpu386    Cpu = 7
46     CpuAmd64 Cpu = Cpu386 + 1<<24
47 )
48
49 var cpuStrings = []intName{
50     {uint32(Cpu386), "Cpu386"},
51     {uint32(CpuAmd64), "CpuAmd64"},
52 }
53
54 func (i Cpu) String() string { return stringName(uint32(i))
55 func (i Cpu) GoString() string { return stringName(uint32(i))
56
57 // A LoadCmd is a Mach-0 load command.
58 type LoadCmd uint32
59
60 const (
61     LoadCmdSegment      LoadCmd = 1
62     LoadCmdSyntab      LoadCmd = 2
63     LoadCmdThread       LoadCmd = 4
64     LoadCmdUnixThread  LoadCmd = 5 // thread+stack
65     LoadCmdDysyntab    LoadCmd = 11
66     LoadCmdDylib       LoadCmd = 12
67     LoadCmdDylinker    LoadCmd = 15
68     LoadCmdSegment64   LoadCmd = 25
69 )
70
71 var cmdStrings = []intName{
72     {uint32(LoadCmdSegment), "LoadCmdSegment"},
73     {uint32(LoadCmdThread), "LoadCmdThread"},
74     {uint32(LoadCmdUnixThread), "LoadCmdUnixThread"},
75     {uint32(LoadCmdDylib), "LoadCmdDylib"},
76     {uint32(LoadCmdSegment64), "LoadCmdSegment64"},
77 }
78
79 func (i LoadCmd) String() string { return stringName(uint32(i))
80 func (i LoadCmd) GoString() string { return stringName(uint32(i))
81
82 // A Segment64 is a 64-bit Mach-0 segment load command.
83 type Segment64 struct {
84     Cmd      LoadCmd
85     Len      uint32
86     Name     [16]byte
87     Addr     uint64
88     Memsz    uint64
89     Offset   uint64
90     Filesz   uint64
91     Maxprot  uint32

```

```

92         Prot    uint32
93         Nsect   uint32
94         Flag    uint32
95     }
96
97     // A Segment32 is a 32-bit Mach-0 segment load command.
98     type Segment32 struct {
99         Cmd      LoadCmd
100        Len      uint32
101        Name     [16]byte
102        Addr     uint32
103        Memsz    uint32
104        Offset   uint32
105        Filesz   uint32
106        Maxprot  uint32
107        Prot     uint32
108        Nsect    uint32
109        Flag     uint32
110    }
111
112     // A DylibCmd is a Mach-0 load dynamic library command.
113     type DylibCmd struct {
114         Cmd      LoadCmd
115         Len      uint32
116         Name     uint32
117         Time     uint32
118         CurrentVersion uint32
119         CompatVersion uint32
120    }
121
122     // A Section32 is a 32-bit Mach-0 section header.
123     type Section32 struct {
124         Name     [16]byte
125         Seg      [16]byte
126         Addr     uint32
127         Size     uint32
128         Offset   uint32
129         Align    uint32
130         Reloff   uint32
131         Nreloc   uint32
132         Flags    uint32
133         Reserve1 uint32
134         Reserve2 uint32
135    }
136
137     // A Section32 is a 64-bit Mach-0 section header.
138     type Section64 struct {
139         Name     [16]byte
140         Seg      [16]byte

```

```

141         Addr      uint64
142         Size      uint64
143         Offset    uint32
144         Align     uint32
145         Reloff    uint32
146         Nreloc    uint32
147         Flags     uint32
148         Reserve1  uint32
149         Reserve2  uint32
150         Reserve3  uint32
151     }
152
153     // A SymtabCmd is a Mach-0 symbol table command.
154     type SymtabCmd struct {
155         Cmd      LoadCmd
156         Len      uint32
157         Symoff   uint32
158         Nsyms   uint32
159         Stroff   uint32
160         Strsize  uint32
161     }
162
163     // A DysymtabCmd is a Mach-0 dynamic symbol table command.
164     type DysymtabCmd struct {
165         Cmd      LoadCmd
166         Len      uint32
167         Ilocalsym uint32
168         Nlocalsym uint32
169         Iextdefsym uint32
170         Nextdefsym uint32
171         Iundefsym  uint32
172         Nundefsym  uint32
173         Tocoffset  uint32
174         Ntoc       uint32
175         Modtaboff  uint32
176         Nmodtab    uint32
177         Extrefsymoff uint32
178         Nextrefsyms uint32
179         Indirectsymoff uint32
180         Nindirectsyms uint32
181         Extreloff  uint32
182         Nextrel    uint32
183         Locreloff  uint32
184         Nlocrel    uint32
185     }
186
187     // An Nlist32 is a Mach-0 32-bit symbol table entry.
188     type Nlist32 struct {
189         Name  uint32

```

```

190         Type  uint8
191         Sect  uint8
192         Desc  uint16
193         Value uint32
194     }
195
196 // An Nlist64 is a Mach-0 64-bit symbol table entry.
197 type Nlist64 struct {
198     Name  uint32
199     Type  uint8
200     Sect  uint8
201     Desc  uint16
202     Value uint64
203 }
204
205 // A Symbol is a Mach-0 32-bit or 64-bit symbol table entry.
206 type Symbol struct {
207     Name  string
208     Type  uint8
209     Sect  uint8
210     Desc  uint16
211     Value uint64
212 }
213
214 // A Thread is a Mach-0 thread state command.
215 type Thread struct {
216     Cmd  LoadCmd
217     Len  uint32
218     Type uint32
219     Data []uint32
220 }
221
222 // Regs386 is the Mach-0 386 register structure.
223 type Regs386 struct {
224     AX  uint32
225     BX  uint32
226     CX  uint32
227     DX  uint32
228     DI  uint32
229     SI  uint32
230     BP  uint32
231     SP  uint32
232     SS  uint32
233     FLAGS uint32
234     IP  uint32
235     CS  uint32
236     DS  uint32
237     ES  uint32
238     FS  uint32
239     GS  uint32

```

```

240 }
241
242 // RegsAMD64 is the Mach-0 AMD64 register structure.
243 type RegsAMD64 struct {
244     AX    uint64
245     BX    uint64
246     CX    uint64
247     DX    uint64
248     DI    uint64
249     SI    uint64
250     BP    uint64
251     SP    uint64
252     R8    uint64
253     R9    uint64
254     R10   uint64
255     R11   uint64
256     R12   uint64
257     R13   uint64
258     R14   uint64
259     R15   uint64
260     IP    uint64
261     FLAGS uint64
262     CS    uint64
263     FS    uint64
264     GS    uint64
265 }
266
267 type intName struct {
268     i uint32
269     s string
270 }
271
272 func stringName(i uint32, names []intName, goSyntax bool) st
273     for _, n := range names {
274         if n.i == i {
275             if goSyntax {
276                 return "macho." + n.s
277             }
278             return n.s
279         }
280     }
281     return strconv.FormatUint(uint64(i), 10)
282 }
283
284 func flagName(i uint32, names []intName, goSyntax bool) stri
285     s := ""
286     for _, n := range names {
287         if n.i&i == n.i {
288             if len(s) > 0 {

```

```
289             s += "+"
290         }
291         if goSyntax {
292             s += "macho."
293         }
294         s += n.s
295         i -= n.i
296     }
297 }
298 if len(s) == 0 {
299     return "0x" + strconv.FormatUint(uint64(i),
300 }
301 if i != 0 {
302     s += "+0x" + strconv.FormatUint(uint64(i), 1
303 }
304 return s
305 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/debug/pe/file.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package pe implements access to PE (Microsoft Windows Por
6 package pe
7
8 import (
9     "debug/dwarf"
10    "encoding/binary"
11    "errors"
12    "fmt"
13    "io"
14    "os"
15    "strconv"
16 )
17
18 // A File represents an open PE file.
19 type File struct {
20     FileHeader
21     Sections []*Section
22
23     closer io.Closer
24 }
25
26 type SectionHeader struct {
27     Name                string
28     VirtualSize         uint32
29     VirtualAddress      uint32
30     Size                 uint32
31     Offset              uint32
32     PointerToRelocations uint32
33     PointerToLineNumbers uint32
34     NumberOfRelocations uint16
35     NumberOfLineNumbers uint16
36     Characteristics     uint32
37 }
38
39 type Section struct {
40     SectionHeader
41
42     // Embed ReaderAt for ReadAt method.
43     // Do not embed SectionReader directly
44     // to avoid having Read and Seek.
```

```

45         // If a client wants Read and Seek it must use
46         // Open() to avoid fighting over the seek offset
47         // with other clients.
48         io.ReaderAt
49         sr *io.SectionReader
50     }
51
52     type ImportDirectory struct {
53         OriginalFirstThunk uint32
54         TimeDateStamp      uint32
55         ForwarderChain      uint32
56         Name                 uint32
57         FirstThunk          uint32
58
59         dll string
60     }
61
62     // Data reads and returns the contents of the PE section.
63     func (s *Section) Data() ([]byte, error) {
64         dat := make([]byte, s.sr.Size())
65         n, err := s.sr.ReadAt(dat, 0)
66         return dat[0:n], err
67     }
68
69     // Open returns a new ReadSeeker reading the PE section.
70     func (s *Section) Open() io.ReadSeeker { return io.NewSectionReader(s.sr, s.offset) }
71
72     type FormatError struct {
73         off int64
74         msg string
75         val interface{}
76     }
77
78     func (e *FormatError) Error() string {
79         msg := e.msg
80         if e.val != nil {
81             msg += fmt.Sprintf(" '%v'", e.val)
82         }
83         msg += fmt.Sprintf(" in record at byte %#x", e.off)
84         return msg
85     }
86
87     // Open opens the named file using os.Open and prepares it for
88     func Open(name string) (*File, error) {
89         f, err := os.Open(name)
90         if err != nil {
91             return nil, err
92         }
93         ff, err := NewFile(f)
94         if err != nil {

```

```

95         f.Close()
96         return nil, err
97     }
98     ff.closer = f
99     return ff, nil
100 }
101
102 // Close closes the File.
103 // If the File was created using NewFile directly instead of
104 // Close has no effect.
105 func (f *File) Close() error {
106     var err error
107     if f.closer != nil {
108         err = f.closer.Close()
109         f.closer = nil
110     }
111     return err
112 }
113
114 // NewFile creates a new File for accessing a PE binary in a
115 func NewFile(r io.ReaderAt) (*File, error) {
116     f := new(File)
117     sr := io.NewSectionReader(r, 0, 1<<63-1)
118
119     var dosheader [96]byte
120     if _, err := r.ReadAt(dosheader[0:], 0); err != nil
121         return nil, err
122     }
123     var base int64
124     if dosheader[0] == 'M' && dosheader[1] == 'Z' {
125         var sign [4]byte
126         r.ReadAt(sign[0:], int64(dosheader[0x3c]))
127         if !(sign[0] == 'P' && sign[1] == 'E' && sig
128             return nil, errors.New("Invalid PE F
129         }
130         base = int64(dosheader[0x3c]) + 4
131     } else {
132         base = int64(0)
133     }
134     sr.Seek(base, os.SEEK_SET)
135     if err := binary.Read(sr, binary.LittleEndian, &f.Fi
136         return nil, err
137     }
138     if f.FileHeader.Machine != IMAGE_FILE_MACHINE_UNKNOW
139         return nil, errors.New("Invalid PE File Form
140     }
141     // get symbol string table
142     sr.Seek(int64(f.FileHeader.PointerToSymbolTable+18*f
143     var l uint32

```

```

144     if err := binary.Read(sr, binary.LittleEndian, &l);
145         return nil, err
146     }
147     ss := make([]byte, l)
148     if _, err := r.ReadAt(ss, int64(f.FileHeader.Pointer
149         return nil, err
150     }
151     sr.Seek(base, os.SEEK_SET)
152     binary.Read(sr, binary.LittleEndian, &f.FileHeader)
153     sr.Seek(int64(f.FileHeader.SizeOfOptionalHeader), os
154     f.Sections = make([]*Section, f.FileHeader.NumberOfS
155     for i := 0; i < int(f.FileHeader.NumberOfSections);
156         sh := new(SectionHeader32)
157         if err := binary.Read(sr, binary.LittleEndia
158             return nil, err
159     }
160     var name string
161     if sh.Name[0] == '\x2F' {
162         si, _ := strconv.Atoi(cstring(sh.Nam
163         name, _ = getString(ss, si)
164     } else {
165         name = cstring(sh.Name[0:])
166     }
167     s := new(Section)
168     s.SectionHeader = SectionHeader{
169         Name:                name,
170         VirtualSize:         uint32(sh.Virt
171         VirtualAddress:    uint32(sh.Virt
172         Size:               uint32(sh.Size
173         Offset:            uint32(sh.Poin
174         PointerToRelocations: uint32(sh.Poin
175         PointerToLineNumbers: uint32(sh.Poin
176         NumberOfRelocations: uint16(sh.Numb
177         NumberOfLineNumbers: uint16(sh.Numb
178         Characteristics:   uint32(sh.Char
179     }
180     s.sr = io.NewSectionReader(r, int64(s.Sectio
181     s.ReaderAt = s.sr
182     f.Sections[i] = s
183 }
184 return f, nil
185 }
186
187 func cstring(b []byte) string {
188     var i int
189     for i = 0; i < len(b) && b[i] != 0; i++ {
190     }
191     return string(b[0:i])
192 }

```

```

193
194 // getString extracts a string from symbol string table.
195 func getString(section []byte, start int) (string, bool) {
196     if start < 0 || start >= len(section) {
197         return "", false
198     }
199
200     for end := start; end < len(section); end++ {
201         if section[end] == 0 {
202             return string(section[start:end]), true
203         }
204     }
205     return "", false
206 }
207
208 // Section returns the first section with the given name, or
209 // section exists.
210 func (f *File) Section(name string) *Section {
211     for _, s := range f.Sections {
212         if s.Name == name {
213             return s
214         }
215     }
216     return nil
217 }
218
219 func (f *File) DWARF() (*dwarf.Data, error) {
220     // There are many other DWARF sections, but these
221     // are the required ones, and the debug/dwarf package
222     // does not use the others, so don't bother loading
223     var names = [...]string{"abbrev", "info", "str"}
224     var dat [len(names)][]byte
225     for i, name := range names {
226         name = ".debug_" + name
227         s := f.Section(name)
228         if s == nil {
229             continue
230         }
231         b, err := s.Data()
232         if err != nil && uint32(len(b)) < s.Size {
233             return nil, err
234         }
235         dat[i] = b
236     }
237
238     abbrev, info, str := dat[0], dat[1], dat[2]
239     return dwarf.New(abbrev, nil, nil, info, nil, nil, nil)
240 }
241
242 // ImportedSymbols returns the names of all symbols

```

```

243 // referred to by the binary f that are expected to be
244 // satisfied by other libraries at dynamic load time.
245 // It does not return weak symbols.
246 func (f *File) ImportedSymbols() ([]string, error) {
247     pe64 := f.Machine == IMAGE_FILE_MACHINE_AMD64
248     ds := f.Section(".idata")
249     if ds == nil {
250         // not dynamic, so no libraries
251         return nil, nil
252     }
253     d, err := ds.Data()
254     if err != nil {
255         return nil, err
256     }
257     var ida []ImportDirectory
258     for len(d) > 0 {
259         var dt ImportDirectory
260         dt.OriginalFirstThunk = binary.LittleEndian.
261         dt.Name = binary.LittleEndian.Uint32(d[12:16])
262         dt.FirstThunk = binary.LittleEndian.Uint32(d[20:24])
263         d = d[24:]
264         if dt.OriginalFirstThunk == 0 {
265             break
266         }
267         ida = append(ida, dt)
268     }
269     names, _ := ds.Data()
270     var all []string
271     for _, dt := range ida {
272         dt.dll, _ = getString(names, int(dt.Name-ds.
273         d, _ = ds.Data()
274         // seek to OriginalFirstThunk
275         d = d[dt.OriginalFirstThunk-ds.VirtualAddress:]
276         for len(d) > 0 {
277             if pe64 { // 64bit
278                 va := binary.LittleEndian.Uint64(d)
279                 d = d[8:]
280                 if va == 0 {
281                     break
282                 }
283                 if va&0x8000000000000000 > 0 {
284                     // TODO add dynimpor
285                 } else {
286                     fn, _ := getString(names, int(va-ds.
287                     all = append(all, fn)
288                 }
289             } else { // 32bit
290                 va := binary.LittleEndian.Uint32(d)
291                 d = d[4:]

```

```

292         if va == 0 {
293             break
294         }
295         if va&0x80000000 > 0 { // is
296             // TODO add dynimpor
297             //ord := va&0x0000FF
298         } else {
299             fn, _ := getString(n
300             all = append(all, fn
301         }
302     }
303 }
304 }
305
306     return all, nil
307 }
308
309 // ImportedLibraries returns the names of all libraries
310 // referred to by the binary f that are expected to be
311 // linked with the binary at dynamic link time.
312 func (f *File) ImportedLibraries() ([]string, error) {
313     // TODO
314     // cgo -dynimport don't use this for windows PE, so
315     return nil, nil
316 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/debug/pe/pe.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package pe
6
7 type FileHeader struct {
8     Machine          uint16
9     NumberOfSections uint16
10    TimeDateStamp    uint32
11    PointerToSymbolTable uint32
12    NumberOfSymbols   uint32
13    SizeOfOptionalHeader uint16
14    Characteristics   uint16
15 }
16
17 type SectionHeader32 struct {
18     Name          [8]uint8
19     VirtualSize   uint32
20     VirtualAddress uint32
21     SizeOfRawData uint32
22     PointerToRawData uint32
23     PointerToRelocations uint32
24     PointerToLineNumbers uint32
25     NumberOfRelocations uint16
26     NumberOfLineNumbers uint16
27     Characteristics   uint32
28 }
29
30 const (
31     IMAGE_FILE_MACHINE_UNKNOWN = 0x0
32     IMAGE_FILE_MACHINE_AM33    = 0x1d3
33     IMAGE_FILE_MACHINE_AMD64   = 0x8664
34     IMAGE_FILE_MACHINE_ARM     = 0x1c0
35     IMAGE_FILE_MACHINE_EBC     = 0xebc
36     IMAGE_FILE_MACHINE_I386    = 0x14c
37     IMAGE_FILE_MACHINE_IA64    = 0x200
38     IMAGE_FILE_MACHINE_M32R    = 0x9041
39     IMAGE_FILE_MACHINE_MIPS16  = 0x266
40     IMAGE_FILE_MACHINE_MIPSFPU = 0x366
41     IMAGE_FILE_MACHINE_MIPSFPU16 = 0x466
42     IMAGE_FILE_MACHINE_POWERPC = 0x1f0
43     IMAGE_FILE_MACHINE_POWERPCFP = 0x1f1
44     IMAGE_FILE_MACHINE_R4000   = 0x166
```

```
45     IMAGE_FILE_MACHINE_SH3           = 0x1a2
46     IMAGE_FILE_MACHINE_SH3DSP        = 0x1a3
47     IMAGE_FILE_MACHINE_SH4           = 0x1a6
48     IMAGE_FILE_MACHINE_SH5           = 0x1a8
49     IMAGE_FILE_MACHINE_THUMB         = 0x1c2
50     IMAGE_FILE_MACHINE_WCEMIPSV2    = 0x169
51 )
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/ascii85/ascii85.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package ascii85 implements the ascii85 data encoding
6 // as used in the btoa tool and Adobe's PostScript and PDF d
7 package ascii85
8
9 import (
10     "io"
11     "strconv"
12 )
13
14 /*
15  * Encoder
16  */
17
18 // Encode encodes src into at most MaxEncodedLen(len(src))
19 // bytes of dst, returning the actual number of bytes writte
20 //
21 // The encoding handles 4-byte chunks, using a special encod
22 // for the last fragment, so Encode is not appropriate for u
23 // individual blocks of a large data stream. Use NewEncoder
24 //
25 // Often, ascii85-encoded data is wrapped in <~ and ~> symbo
26 // Encode does not add these.
27 func Encode(dst, src []byte) int {
28     if len(src) == 0 {
29         return 0
30     }
31
32     n := 0
33     for len(src) > 0 {
34         dst[n] = 0
35         dst[n+1] = 0
36         dst[n+2] = 0
37         dst[n+3] = 0
38         dst[n+4] = 0
39
40         // Unpack 4 bytes into uint32 to repack into
41         var v uint32
```

```

42         switch len(src) {
43         default:
44             v |= uint32(src[3])
45             fallthrough
46         case 3:
47             v |= uint32(src[2]) << 8
48             fallthrough
49         case 2:
50             v |= uint32(src[1]) << 16
51             fallthrough
52         case 1:
53             v |= uint32(src[0]) << 24
54         }
55
56         // Special case: zero (!!!!!) shortens to z.
57         if v == 0 && len(src) >= 4 {
58             dst[0] = 'z'
59             dst = dst[1:]
60             src = src[4:]
61             n++
62             continue
63         }
64
65         // Otherwise, 5 base 85 digits starting at !
66         for i := 4; i >= 0; i-- {
67             dst[i] = '!' + byte(v%85)
68             v /= 85
69         }
70
71         // If src was short, discard the low destina
72         m := 5
73         if len(src) < 4 {
74             m -= 4 - len(src)
75             src = nil
76         } else {
77             src = src[4:]
78         }
79         dst = dst[m:]
80         n += m
81     }
82     return n
83 }
84
85 // MaxEncodedLen returns the maximum length of an encoding o
86 func MaxEncodedLen(n int) int { return (n + 3) / 4 * 5 }
87
88 // NewEncoder returns a new ascii85 stream encoder. Data wr
89 // the returned writer will be encoded and then written to w
90 // Ascii85 encodings operate in 32-bit blocks; when finished
91 // writing, the caller must Close the returned encoder to fl

```

```

92 // trailing partial block.
93 func NewEncoder(w io.Writer) io.WriteCloser { return &encode
94
95 type encoder struct {
96     err error
97     w   io.Writer
98     buf [4]byte // buffered data waiting to be encoded
99     nbuf int // number of bytes in buf
100    out [1024]byte // output buffer
101 }
102
103 func (e *encoder) Write(p []byte) (n int, err error) {
104     if e.err != nil {
105         return 0, e.err
106     }
107
108     // Leading fringe.
109     if e.nbuf > 0 {
110         var i int
111         for i = 0; i < len(p) && e.nbuf < 4; i++ {
112             e.buf[e.nbuf] = p[i]
113             e.nbuf++
114         }
115         n += i
116         p = p[i:]
117         if e.nbuf < 4 {
118             return
119         }
120         nout := Encode(e.out[0:], e.buf[0:])
121         if _, e.err = e.w.Write(e.out[0:nout]); e.err
122             return n, e.err
123     }
124     e.nbuf = 0
125 }
126
127 // Large interior chunks.
128 for len(p) >= 4 {
129     nn := len(e.out) / 5 * 4
130     if nn > len(p) {
131         nn = len(p)
132     }
133     nn -= nn % 4
134     if nn > 0 {
135         nout := Encode(e.out[0:], p[0:nn])
136         if _, e.err = e.w.Write(e.out[0:nout]); e.err
137             return n, e.err
138     }
139 }
140 n += nn

```

```

141         p = p[nn:]
142     }
143
144     // Trailing fringe.
145     for i := 0; i < len(p); i++ {
146         e.buf[i] = p[i]
147     }
148     e.nbuf = len(p)
149     n += len(p)
150     return
151 }
152
153 // Close flushes any pending output from the encoder.
154 // It is an error to call Write after calling Close.
155 func (e *encoder) Close() error {
156     // If there's anything left in the buffer, flush it
157     if e.err == nil && e.nbuf > 0 {
158         nout := Encode(e.out[0:], e.buf[0:e.nbuf])
159         e.nbuf = 0
160         _, e.err = e.w.Write(e.out[0:nout])
161     }
162     return e.err
163 }
164
165 /*
166  * Decoder
167  */
168
169 type CorruptInputError int64
170
171 func (e CorruptInputError) Error() string {
172     return "illegal ascii85 data at input byte " + strconv.
173 }
174
175 // Decode decodes src into dst, returning both the number
176 // of bytes written to dst and the number consumed from src.
177 // If src contains invalid ascii85 data, Decode will return
178 // number of bytes successfully written and a CorruptInputEr
179 // Decode ignores space and control characters in src.
180 // Often, ascii85-encoded data is wrapped in <~ and ~> symbo
181 // Decode expects these to have been stripped by the caller.
182 //
183 // If flush is true, Decode assumes that src represents the
184 // end of the input stream and processes it completely rathe
185 // than wait for the completion of another 32-bit block.
186 //
187 // NewDecoder wraps an io.Reader interface around Decode.
188 //
189 func Decode(dst, src []byte, flush bool) (ndst, nsrc int, er

```

```

190     var v uint32
191     var nb int
192     for i, b := range src {
193         if len(dst)-ndst < 4 {
194             return
195         }
196         switch {
197         case b <= ' ':
198             continue
199         case b == 'z' && nb == 0:
200             nb = 5
201             v = 0
202         case '!' <= b && b <= 'u':
203             v = v*85 + uint32(b-'!')
204             nb++
205         default:
206             return 0, 0, CorruptInputError(i)
207         }
208         if nb == 5 {
209             nsrc = i + 1
210             dst[ndst] = byte(v >> 24)
211             dst[ndst+1] = byte(v >> 16)
212             dst[ndst+2] = byte(v >> 8)
213             dst[ndst+3] = byte(v)
214             ndst += 4
215             nb = 0
216             v = 0
217         }
218     }
219     if flush {
220         nsrc = len(src)
221         if nb > 0 {
222             // The number of output bytes in the
223             // is the number of leftover input b
224             // the extra byte provides enough bi
225             // the inefficiency of the encoding
226             if nb == 1 {
227                 return 0, 0, CorruptInputErr
228             }
229             for i := nb; i < 5; i++ {
230                 // The short encoding trunca
231                 // We have to assume the wor
232                 // in order to ensure that t
233                 v = v*85 + 84
234             }
235             for i := 0; i < nb-1; i++ {
236                 dst[ndst] = byte(v >> 24)
237                 v <<= 8
238                 ndst++
239             }

```

```

240         }
241     }
242     return
243 }
244
245 // NewDecoder constructs a new ascii85 stream decoder.
246 func NewDecoder(r io.Reader) io.Reader { return &decoder{r:
247
248 type decoder struct {
249     err      error
250     readErr  error
251     r        io.Reader
252     end      bool        // saw end of message
253     buf      [1024]byte // leftover input
254     nbuf     int
255     out      []byte     // leftover decoded output
256     outbuf   [1024]byte
257 }
258
259 func (d *decoder) Read(p []byte) (n int, err error) {
260     if len(p) == 0 {
261         return 0, nil
262     }
263     if d.err != nil {
264         return 0, d.err
265     }
266
267     for {
268         // Copy leftover output from last decode.
269         if len(d.out) > 0 {
270             n = copy(p, d.out)
271             d.out = d.out[n:]
272             return
273         }
274
275         // Decode leftover input from last read.
276         var nn, nsrc, ndst int
277         if d.nbuf > 0 {
278             ndst, nsrc, d.err = Decode(d.outbuf[
279                 if ndst > 0 {
280                     d.out = d.outbuf[0:ndst]
281                     d.nbuf = copy(d.buf[0:], d.b
282                         continue // copy out and ret
283                 }
284             }
285
286         // Out of input, out of decoded output. Che
287         if d.err != nil {
288             return 0, d.err

```

```
289         }
290         if d.readErr != nil {
291             d.err = d.readErr
292             return 0, d.err
293         }
294
295         // Read more data.
296         nn, d.readErr = d.r.Read(d.buf[d.nbuf:])
297         d.nbuf += nn
298     }
299     panic("unreachable")
300 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/asn1/asn1.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package asn1 implements parsing of DER-encoded ASN.1 data
6 // as defined in ITU-T Rec X.690.
7 //
8 // See also ``A Layman's Guide to a Subset of ASN.1, BER, and
9 // http://luca.ntop.org/Teaching/Appunti/asn1.html.
10 package asn1
11
12 // ASN.1 is a syntax for specifying abstract objects and BER
13 // are different encoding formats for those objects. Here, w
14 // with DER, the Distinguished Encoding Rules. DER is used i
15 // it's fast to parse and, unlike BER, has a unique encoding
16 // When calculating hashes over objects, it's important that
17 // bytes be the same at both ends and DER removes this margi
18 //
19 // ASN.1 is very complex and this package doesn't attempt to
20 // everything by any means.
21
22 import (
23     "fmt"
24     "math/big"
25     "reflect"
26     "time"
27 )
28
29 // A StructuralError suggests that the ASN.1 data is valid,
30 // which is receiving it doesn't match.
31 type StructuralError struct {
32     Msg string
33 }
34
35 func (e StructuralError) Error() string { return "ASN.1 stru
36
37 // A SyntaxError suggests that the ASN.1 data is invalid.
38 type SyntaxError struct {
39     Msg string
40 }
41
```

```

42 func (e SyntaxError) Error() string { return "ASN.1 syntax e
43
44 // We start by dealing with each of the primitive types in t
45
46 // BOOLEAN
47
48 func parseBool(bytes []byte) (ret bool, err error) {
49     if len(bytes) != 1 {
50         err = SyntaxError{"invalid boolean"}
51         return
52     }
53
54     return bytes[0] != 0, nil
55 }
56
57 // INTEGER
58
59 // parseInt64 treats the given bytes as a big-endian, signed
60 // returns the result.
61 func parseInt64(bytes []byte) (ret int64, err error) {
62     if len(bytes) > 8 {
63         // We'll overflow an int64 in this case.
64         err = StructuralError{"integer too large"}
65         return
66     }
67     for bytesRead := 0; bytesRead < len(bytes); bytesRea
68         ret <<= 8
69         ret |= int64(bytes[bytesRead])
70     }
71
72     // Shift up and down in order to sign extend the res
73     ret <<= 64 - uint8(len(bytes))*8
74     ret >>= 64 - uint8(len(bytes))*8
75     return
76 }
77
78 // parseInt treats the given bytes as a big-endian, signed i
79 // the result.
80 func parseInt(bytes []byte) (int, error) {
81     ret64, err := parseInt64(bytes)
82     if err != nil {
83         return 0, err
84     }
85     if ret64 != int64(int(ret64)) {
86         return 0, StructuralError{"integer too large
87     }
88     return int(ret64), nil
89 }
90
91 var bigOne = big.NewInt(1)

```

```

92
93 // parseBigInt treats the given bytes as a big-endian, signe
94 // the result.
95 func parseBigInt(bytes []byte) *big.Int {
96     ret := new(big.Int)
97     if len(bytes) > 0 && bytes[0]&0x80 == 0x80 {
98         // This is a negative number.
99         notBytes := make([]byte, len(bytes))
100        for i := range notBytes {
101            notBytes[i] = ^bytes[i]
102        }
103        ret.SetBytes(notBytes)
104        ret.Add(ret, bigOne)
105        ret.Neg(ret)
106        return ret
107    }
108    ret.SetBytes(bytes)
109    return ret
110 }
111
112 // BIT STRING
113
114 // BitString is the structure to use when you want an ASN.1
115 // bit string is padded up to the nearest byte in memory and
116 // valid bits is recorded. Padding bits will be zero.
117 type BitString struct {
118     Bytes      []byte // bits packed into bytes.
119     BitLength int    // length in bits.
120 }
121
122 // At returns the bit at the given index. If the index is ou
123 // returns false.
124 func (b BitString) At(i int) int {
125     if i < 0 || i >= b.BitLength {
126         return 0
127     }
128     x := i / 8
129     y := 7 - uint(i%8)
130     return int(b.Bytes[x]>>y) & 1
131 }
132
133 // RightAlign returns a slice where the padding bits are at
134 // slice may share memory with the BitString.
135 func (b BitString) RightAlign() []byte {
136     shift := uint(8 - (b.BitLength % 8))
137     if shift == 8 || len(b.Bytes) == 0 {
138         return b.Bytes
139     }
140

```

```

141     a := make([]byte, len(b.Bytes))
142     a[0] = b.Bytes[0] >> shift
143     for i := 1; i < len(b.Bytes); i++ {
144         a[i] = b.Bytes[i-1] << (8 - shift)
145         a[i] |= b.Bytes[i] >> shift
146     }
147
148     return a
149 }
150
151 // parseBitString parses an ASN.1 bit string from the given
152 func parseBitString(bytes []byte) (ret BitString, err error)
153     if len(bytes) == 0 {
154         err = SyntaxError{"zero length BIT STRING"}
155         return
156     }
157     paddingBits := int(bytes[0])
158     if paddingBits > 7 ||
159         len(bytes) == 1 && paddingBits > 0 ||
160         bytes[len(bytes)-1]&((1<<bytes[0])-1) != 0 {
161         err = SyntaxError{"invalid padding bits in B"}
162         return
163     }
164     ret.BitLength = (len(bytes)-1)*8 - paddingBits
165     ret.Bytes = bytes[1:]
166     return
167 }
168
169 // OBJECT IDENTIFIER
170
171 // An ObjectIdentifier represents an ASN.1 OBJECT IDENTIFIER
172 type ObjectIdentifier []int
173
174 // Equal returns true iff oi and other represent the same id
175 func (oi ObjectIdentifier) Equal(other ObjectIdentifier) bool
176     if len(oi) != len(other) {
177         return false
178     }
179     for i := 0; i < len(oi); i++ {
180         if oi[i] != other[i] {
181             return false
182         }
183     }
184
185     return true
186 }
187
188 // parseObjectIdentifier parses an OBJECT IDENTIFIER from th
189 // returns it. An object identifier is a sequence of variabl

```

```

190 // that are assigned in a hierarchy.
191 func parseObjectIdentifier(bytes []byte) (s []int, err error) {
192     if len(bytes) == 0 {
193         err = SyntaxError{"zero length OBJECT IDENTIFI
194         return
195     }
196
197     // In the worst case, we get two elements from the f
198     // encoded differently) and then every varint is a s
199     s = make([]int, len(bytes)+1)
200
201     // The first byte is 40*value1 + value2:
202     s[0] = int(bytes[0]) / 40
203     s[1] = int(bytes[0]) % 40
204     i := 2
205     for offset := 1; offset < len(bytes); i++ {
206         var v int
207         v, offset, err = parseBase128Int(bytes, offs
208         if err != nil {
209             return
210         }
211         s[i] = v
212     }
213     s = s[0:i]
214     return
215 }
216
217 // ENUMERATED
218
219 // An Enumerated is represented as a plain int.
220 type Enumerated int
221
222 // FLAG
223
224 // A Flag accepts any data and is set to true if present.
225 type Flag bool
226
227 // parseBase128Int parses a base-128 encoded int from the gi
228 // given byte slice. It returns the value and the new offset
229 func parseBase128Int(bytes []byte, initOffset int) (ret, off
230     offset = initOffset
231     for shifted := 0; offset < len(bytes); shifted++ {
232         if shifted > 4 {
233             err = StructuralError{"base 128 inte
234             return
235         }
236         ret <<= 7
237         b := bytes[offset]
238         ret |= int(b & 0x7f)
239         offset++

```

```

240             if b&0x80 == 0 {
241                 return
242             }
243         }
244         err = SyntaxError{"truncated base 128 integer"}
245         return
246     }
247
248     // UTCTime
249
250     func parseUTCTime(bytes []byte) (ret time.Time, err error) {
251         s := string(bytes)
252         ret, err = time.Parse("0601021504Z0700", s)
253         if err != nil {
254             ret, err = time.Parse("060102150405Z0700", s)
255         }
256         if err == nil && ret.Year() >= 2050 {
257             // UTCTime only encodes times prior to 2050.
258             ret = ret.AddDate(-100, 0, 0)
259         }
260
261         return
262     }
263
264     // parseGeneralizedTime parses the GeneralizedTime from the
265     // and returns the resulting time.
266     func parseGeneralizedTime(bytes []byte) (ret time.Time, err
267         return time.Parse("20060102150405Z0700", string(byte
268     }
269
270     // PrintableString
271
272     // parsePrintableString parses a ASN.1 PrintableString from
273     // array and returns it.
274     func parsePrintableString(bytes []byte) (ret string, err err
275         for _, b := range bytes {
276             if !isPrintable(b) {
277                 err = SyntaxError{"PrintableString c
278                 return
279             }
280         }
281         ret = string(bytes)
282         return
283     }
284
285     // isPrintable returns true iff the given b is in the ASN.1
286     func isPrintable(b byte) bool {
287         return 'a' <= b && b <= 'z' ||
288             'A' <= b && b <= 'Z' ||

```

```

289         '0' <= b && b <= '9' ||
290         '\ ' <= b && b <= ')' ||
291         '+' <= b && b <= '/' ||
292         b == ' ' ||
293         b == ':' ||
294         b == '=' ||
295         b == '?' ||
296         // This is technically not allowed in a Prin
297         // However, x509 certificates with wildcard
298         // always use the correct string type so we
299         b == '*'
300     }
301
302     // IA5String
303
304     // parseIA5String parses a ASN.1 IA5String (ASCII string) fr
305     // byte slice and returns it.
306     func parseIA5String(bytes []byte) (ret string, err error) {
307         for _, b := range bytes {
308             if b >= 0x80 {
309                 err = SyntaxError{"IA5String contain
310                 return
311             }
312         }
313         ret = string(bytes)
314         return
315     }
316
317     // T61String
318
319     // parseT61String parses a ASN.1 T61String (8-bit clean stri
320     // byte slice and returns it.
321     func parseT61String(bytes []byte) (ret string, err error) {
322         return string(bytes), nil
323     }
324
325     // UTF8String
326
327     // parseUTF8String parses a ASN.1 UTF8String (raw UTF-8) fro
328     // array and returns it.
329     func parseUTF8String(bytes []byte) (ret string, err error) {
330         return string(bytes), nil
331     }
332
333     // A RawValue represents an undecoded ASN.1 object.
334     type RawValue struct {
335         Class, Tag int
336         IsCompound bool
337         Bytes      []byte

```

```

338         FullBytes []byte // includes the tag and length
339     }
340
341     // RawContent is used to signal that the undecoded, DER data
342     // preserved for a struct. To use it, the first field of the
343     // this type. It's an error for any of the other fields to h
344     type RawContent []byte
345
346     // Tagging
347
348     // parseTagAndLength parses an ASN.1 tag and length pair fro
349     // into a byte slice. It returns the parsed data and the new
350     // SET OF (tag 17) are mapped to SEQUENCE and SEQUENCE OF (t
351     // don't distinguish between ordered and unordered objects i
352     func parseTagAndLength(bytes []byte, initOffset int) (ret ta
353         offset = initOffset
354         b := bytes[offset]
355         offset++
356         ret.class = int(b >> 6)
357         ret.isCompound = b&0x20 == 0x20
358         ret.tag = int(b & 0x1f)
359
360         // If the bottom five bits are set, then the tag num
361         // encoded afterwards
362         if ret.tag == 0x1f {
363             ret.tag, offset, err = parseBase128Int(bytes
364                 if err != nil {
365                     return
366                 }
367         }
368         if offset >= len(bytes) {
369             err = SyntaxError{"truncated tag or length"}
370             return
371         }
372         b = bytes[offset]
373         offset++
374         if b&0x80 == 0 {
375             // The length is encoded in the bottom 7 bit
376             ret.length = int(b & 0x7f)
377         } else {
378             // Bottom 7 bits give the number of length b
379             numBytes := int(b & 0x7f)
380             if numBytes == 0 {
381                 err = SyntaxError{"indefinite length
382                     return
383             }
384             ret.length = 0
385             for i := 0; i < numBytes; i++ {
386                 if offset >= len(bytes) {
387                     err = SyntaxError{"truncated

```

```

388             return
389         }
390         b = bytes[offset]
391         offset++
392         if ret.length >= 1<<23 {
393             // We can't shift ret.length
394             // overflowing.
395             err = StructuralError{"length
396             return
397         }
398         ret.length <= 8
399         ret.length |= int(b)
400         if ret.length == 0 {
401             // DER requires that lengths
402             err = StructuralError{"super
403             return
404         }
405     }
406 }
407
408     return
409 }
410
411 // parseSequenceOf is used for SEQUENCE OF and SET OF values
412 // a number of ASN.1 values from the given byte slice and re
413 // slice of Go values of the given type.
414 func parseSequenceOf(bytes []byte, sliceType reflect.Type, e
415     expectedTag, compoundType, ok := getUniversalType(e)
416     if !ok {
417         err = StructuralError{"unknown Go type for s
418         return
419     }
420
421     // First we iterate over the input and count the num
422     // checking that the types are correct in each case.
423     numElements := 0
424     for offset := 0; offset < len(bytes); {
425         var t tagAndLength
426         t, offset, err = parseTagAndLength(bytes, of
427         if err != nil {
428             return
429         }
430         // We pretend that GENERAL STRINGS are PRINT
431         // that a sequence of them can be parsed int
432         if t.tag == tagGeneralString {
433             t.tag = tagPrintableString
434         }
435         if t.class != classUniversal || t.isCompound
436         err = StructuralError{"sequence tag

```

```

437             return
438         }
439         if invalidLength(offset, t.length, len(bytes)
440             err = SyntaxError{"truncated sequenc
441             return
442         }
443         offset += t.length
444         numElements++
445     }
446     ret = reflect.MakeSlice(sliceType, numElements, numE
447     params := fieldParameters{}
448     offset := 0
449     for i := 0; i < numElements; i++ {
450         offset, err = parseField(ret.Index(i), bytes
451         if err != nil {
452             return
453         }
454     }
455     return
456 }
457
458 var (
459     bitStringType      = reflect.TypeOf(BitString{})
460     objectIdentifierType = reflect.TypeOf(ObjectIdentifi
461     enumeratedType     = reflect.TypeOf(Enumerated(0))
462     flagType           = reflect.TypeOf(Flag(false))
463     timeType           = reflect.TypeOf(time.Time{})
464     rawValueType       = reflect.TypeOf(RawValue{})
465     rawContentsType    = reflect.TypeOf(RawContent(nil
466     bigIntType         = reflect.TypeOf(new(big.Int))
467 )
468
469 // invalidLength returns true iff offset + length > sliceLen
470 // addition would overflow.
471 func invalidLength(offset, length, sliceLength int) bool {
472     return offset+length < offset || offset+length > sli
473 }
474
475 // parseField is the main parsing function. Given a byte sli
476 // into the array, it will try to parse a suitable ASN.1 val
477 // in the given Value.
478 func parseField(v reflect.Value, bytes []byte, initOffset in
479     offset = initOffset
480     fieldType := v.Type()
481
482     // If we have run out of data, it may be that there
483     if offset == len(bytes) {
484         if !setDefaultValue(v, params) {
485             err = SyntaxError{"sequence truncate

```

```

486         }
487         return
488     }
489
490     // Deal with raw values.
491     if fieldType == rawValueType {
492         var t tagAndLength
493         t, offset, err = parseTagAndLength(bytes, of
494         if err != nil {
495             return
496         }
497         if invalidLength(offset, t.length, len(bytes
498             err = SyntaxError{"data truncated"}
499             return
500         }
501         result := RawValue{t.class, t.tag, t.isCompo
502         offset += t.length
503         v.Set(reflect.ValueOf(result))
504         return
505     }
506
507     // Deal with the ANY type.
508     if ifaceType := fieldType; ifaceType.Kind() == refle
509         var t tagAndLength
510         t, offset, err = parseTagAndLength(bytes, of
511         if err != nil {
512             return
513         }
514         if invalidLength(offset, t.length, len(bytes
515             err = SyntaxError{"data truncated"}
516             return
517         }
518         var result interface{}
519         if !t.isCompound && t.class == classUniversa
520             innerBytes := bytes[offset : offset+
521             switch t.tag {
522             case tagPrintableString:
523                 result, err = parsePrintable
524             case tagIA5String:
525                 result, err = parseIA5String
526             case tagT61String:
527                 result, err = parseT61String
528             case tagUTF8String:
529                 result, err = parseUTF8Strin
530             case tagInteger:
531                 result, err = parseInt64(inn
532             case tagBitString:
533                 result, err = parseBitString
534             case tagOID:
535                 result, err = parseObjectIde

```

```

536         case tagUTCTime:
537             result, err = parseUTCTime(i
538         case tagOctetString:
539             result = innerBytes
540         default:
541             // If we don't know how to h
542         }
543     }
544     offset += t.length
545     if err != nil {
546         return
547     }
548     if result != nil {
549         v.Set(reflect.ValueOf(result))
550     }
551     return
552 }
553 universalTag, compoundType, ok1 := getUniversalType(
554 if !ok1 {
555     err = StructuralError{fmt.Sprintf("unknown G
556     return
557 }
558
559 t, offset, err := parseTagAndLength(bytes, offset)
560 if err != nil {
561     return
562 }
563 if params.explicit {
564     expectedClass := classContextSpecific
565     if params.application {
566         expectedClass = classApplication
567     }
568     if t.class == expectedClass && t.tag == *par
569         if t.length > 0 {
570             t, offset, err = parseTagAnd
571             if err != nil {
572                 return
573             }
574         } else {
575             if fieldType != flagType {
576                 err = StructuralErro
577                 return
578             }
579             v.SetBool(true)
580             return
581         }
582     } else {
583         // The tags didn't match, it might b
584         ok := setDefaultValue(v, params)

```

```

585         if ok {
586             offset = initOffset
587         } else {
588             err = StructuralError{"expli
589         }
590         return
591     }
592 }
593
594 // Special case for strings: all the ASN.1 string ty
595 // type string. getUniversalType returns the tag for
596 // when it sees a string, so if we see a different s
597 // wire, we change the universal type to match.
598 if universalTag == tagPrintableString {
599     switch t.tag {
600     case tagIA5String, tagGeneralString, tagT61S
601         universalTag = t.tag
602     }
603 }
604
605 // Special case for time: UTCTime and GeneralizedTim
606 // Go type time.Time.
607 if universalTag == tagUTCTime && t.tag == tagGeneral
608     universalTag = tagGeneralizedTime
609 }
610
611 expectedClass := classUniversal
612 expectedTag := universalTag
613
614 if !params.explicit && params.tag != nil {
615     expectedClass = classContextSpecific
616     expectedTag = *params.tag
617 }
618
619 if !params.explicit && params.application && params.
620     expectedClass = classApplication
621     expectedTag = *params.tag
622 }
623
624 // We have unwrapped any explicit tagging at this po
625 if t.class != expectedClass || t.tag != expectedTag
626     // Tags don't match. Again, it could be an o
627     ok := setDefaultValue(v, params)
628     if ok {
629         offset = initOffset
630     } else {
631         err = StructuralError{fmt.Sprintf("t
632     }
633     return

```

```

634     }
635     if invalidLength(offset, t.length, len(bytes)) {
636         err = SyntaxError{"data truncated"}
637         return
638     }
639     innerBytes := bytes[offset : offset+t.length]
640     offset += t.length
641
642     // We deal with the structures defined in this packa
643     switch fieldType {
644     case objectIdentifierType:
645         newSlice, err1 := parseObjectIdentifier(inne
646         v.Set(reflect.MakeSlice(v.Type(), len(newSli
647         if err1 == nil {
648             reflect.Copy(v, reflect.ValueOf(newS
649         }
650         err = err1
651         return
652     case bitStringType:
653         bs, err1 := parseBitString(innerBytes)
654         if err1 == nil {
655             v.Set(reflect.ValueOf(bs))
656         }
657         err = err1
658         return
659     case timeType:
660         var time time.Time
661         var err1 error
662         if universalTag == tagUTCTime {
663             time, err1 = parseUTCTime(innerBytes
664         } else {
665             time, err1 = parseGeneralizedTime(in
666         }
667         if err1 == nil {
668             v.Set(reflect.ValueOf(time))
669         }
670         err = err1
671         return
672     case enumeratedType:
673         parsedInt, err1 := parseInt(innerBytes)
674         if err1 == nil {
675             v.SetInt(int64(parsedInt))
676         }
677         err = err1
678         return
679     case flagType:
680         v.SetBool(true)
681         return
682     case bigIntType:
683         parsedInt := parseBigInt(innerBytes)

```

```

684         v.Set(reflect.ValueOf(parsedInt))
685         return
686     }
687     switch val := v; val.Kind() {
688     case reflect.Bool:
689         parsedBool, err1 := parseBool(innerBytes)
690         if err1 == nil {
691             val.SetBool(parsedBool)
692         }
693         err = err1
694         return
695     case reflect.Int, reflect.Int32:
696         parsedInt, err1 := parseInt(innerBytes)
697         if err1 == nil {
698             val.SetInt(int64(parsedInt))
699         }
700         err = err1
701         return
702     case reflect.Int64:
703         parsedInt, err1 := parseInt64(innerBytes)
704         if err1 == nil {
705             val.SetInt(parsedInt)
706         }
707         err = err1
708         return
709     // TODO(dfc) Add support for the remaining integer t
710     case reflect.Struct:
711         structType := fieldType
712
713         if structType.NumField() > 0 &&
714             structType.Field(0).Type == rawConte
715             bytes := bytes[initOffset:offset]
716             val.Field(0).Set(reflect.ValueOf(Raw
717         }
718
719         innerOffset := 0
720         for i := 0; i < structType.NumField(); i++ {
721             field := structType.Field(i)
722             if i == 0 && field.Type == rawConten
723                 continue
724             }
725             innerOffset, err = parseField(val.Fi
726             if err != nil {
727                 return
728             }
729         }
730         // We allow extra bytes at the end of the SE
731         // adding elements to the end has been used
732         // version numbers have increased.

```

```

733         return
734     case reflect.Slice:
735         sliceType := fieldType
736         if sliceType.Elem().Kind() == reflect.Uint8
737             val.Set(reflect.MakeSlice(sliceType,
738                 reflect.Copy(val, reflect.ValueOf(in
739                     return
740             }
741         newSlice, err1 := parseSequenceOf(innerBytes
742         if err1 == nil {
743             val.Set(newSlice)
744         }
745         err = err1
746         return
747     case reflect.String:
748         var v string
749         switch universalTag {
750         case tagPrintableString:
751             v, err = parsePrintableString(innerB
752         case tagIA5String:
753             v, err = parseIA5String(innerBytes)
754         case tagT61String:
755             v, err = parseT61String(innerBytes)
756         case tagUTF8String:
757             v, err = parseUTF8String(innerBytes)
758         case tagGeneralString:
759             // GeneralString is specified in ISO
760             // A brief review suggests that it i
761             // that allow the encoding to change
762             // such. We give up and pass it as a
763             v, err = parseT61String(innerBytes)
764         default:
765             err = SyntaxError{fmt.Sprintf("inter
766         }
767         if err == nil {
768             val.SetString(v)
769         }
770         return
771     }
772     err = StructuralError{"unsupported: " + v.Type().Str
773     return
774 }
775
776 // setDefaultValue is used to install a default value, from
777 // a Value. It is successful is the field was optional, even
778 // wasn't provided or it failed to install it into the Value
779 func setDefaultValue(v reflect.Value, params fieldParameters
780     if !params.optional {
781         return

```

```

782     }
783     ok = true
784     if params.defaultValue == nil {
785         return
786     }
787     switch val := v; val.Kind() {
788     case reflect.Int, reflect.Int8, reflect.Int16, refle
789         val.SetInt(*params.defaultValue)
790     }
791     return
792 }
793
794 // Unmarshal parses the DER-encoded ASN.1 data structure b
795 // and uses the reflect package to fill in an arbitrary valu
796 // Because Unmarshal uses the reflect package, the structs
797 // being written to must use upper case field names.
798 //
799 // An ASN.1 INTEGER can be written to an int, int32, int64,
800 // or *big.Int (from the math/big package).
801 // If the encoded value does not fit in the Go type,
802 // Unmarshal returns a parse error.
803 //
804 // An ASN.1 BIT STRING can be written to a BitString.
805 //
806 // An ASN.1 OCTET STRING can be written to a []byte.
807 //
808 // An ASN.1 OBJECT IDENTIFIER can be written to an
809 // ObjectIdentifier.
810 //
811 // An ASN.1 ENUMERATED can be written to an Enumerated.
812 //
813 // An ASN.1 UTCTIME or GENERALIZEDTIME can be written to a t
814 //
815 // An ASN.1 PrintableString or IA5String can be written to a
816 //
817 // Any of the above ASN.1 values can be written to an interf
818 // The value stored in the interface has the corresponding G
819 // For integers, that type is int64.
820 //
821 // An ASN.1 SEQUENCE OF x or SET OF x can be written
822 // to a slice if an x can be written to the slice's element
823 //
824 // An ASN.1 SEQUENCE or SET can be written to a struct
825 // if each of the elements in the sequence can be
826 // written to the corresponding element in the struct.
827 //
828 // The following tags on struct fields have special meaning
829 //
830 //     optional                marks the field as ASN.1 OPT
831 //     [explicit] tag:x        specifies the ASN.1 tag numb

```

```

832 //      default:x                sets the default value for c
833 //
834 // If the type of the first field of a structure is RawConte
835 // ASN1 contents of the struct will be stored in it.
836 //
837 // Other ASN.1 types are not supported; if it encounters the
838 // Unmarshal returns a parse error.
839 func Unmarshal(b []byte, val interface{}) (rest []byte, err
840             return UnmarshalWithParams(b, val, "")
841 }
842
843 // UnmarshalWithParams allows field parameters to be specifi
844 // top-level element. The form of the params is the same as
845 func UnmarshalWithParams(b []byte, val interface{}, params s
846     v := reflect.ValueOf(val).Elem()
847     offset, err := parseField(v, b, 0, parseFieldParamet
848     if err != nil {
849         return nil, err
850     }
851     return b[offset:], nil
852 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/asn1/common.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package asn1
6
7 import (
8     "reflect"
9     "strconv"
10    "strings"
11 )
12
13 // ASN.1 objects have metadata preceding them:
14 //   the tag: the type of the object
15 //   a flag denoting if this object is compound or not
16 //   the class type: the namespace of the tag
17 //   the length of the object, in bytes
18
19 // Here are some standard tags and classes
20
21 const (
22     tagBoolean          = 1
23     tagInteger          = 2
24     tagBitString        = 3
25     tagOctetString      = 4
26     tagOID              = 6
27     tagEnum             = 10
28     tagUTF8String       = 12
29     tagSequence         = 16
30     tagSet              = 17
31     tagPrintableString  = 19
32     tagT61String        = 20
33     tagIA5String        = 22
34     tagUTCTime          = 23
35     tagGeneralizedTime  = 24
36     tagGeneralString    = 27
37 )
38
39 const (
40     classUniversal      = 0
41     classApplication    = 1
```

```

42         classContextSpecific = 2
43         classPrivate          = 3
44     )
45
46     type tagAndLength struct {
47         class, tag, length int
48         isCompound          bool
49     }
50
51     // ASN.1 has IMPLICIT and EXPLICIT tags, which can be transl
52     // of" and "in addition to". When not specified, every primi
53     // default tag in the UNIVERSAL class.
54     //
55     // For example: a BIT STRING is tagged [UNIVERSAL 3] by defa
56     // doesn't actually have a UNIVERSAL keyword). However, by s
57     // CONTEXT-SPECIFIC 42], that means that the tag is replaced
58     //
59     // On the other hand, if it said [EXPLICIT CONTEXT-SPECIFIC
60     // /additional/ tag would wrap the default tag. This explici
61     // compound flag set.
62     //
63     // (This is used in order to remove ambiguity with optional
64     //
65     // You can layer EXPLICIT and IMPLICIT tags to an arbitrary
66     // don't support that here. We support a single layer of EXP
67     // tagging with tag strings on the fields of a structure.
68
69     // fieldParameters is the parsed representation of tag strin
70     type fieldParameters struct {
71         optional      bool    // true iff the field is OPTIONA
72         explicit      bool    // true iff an EXPLICIT tag is i
73         application   bool    // true iff an APPLICATION tag i
74         defaultValue *int64 // a default value for INTEGER t
75         tag           *int    // the EXPLICIT or IMPLICIT tag
76         stringType   int     // the string tag to use when ma
77         set          bool    // true iff this should be encod
78         omitEmpty    bool    // true iff this should be omitt
79
80         // Invariants:
81         //   if explicit is set, tag is non-nil.
82     }
83
84     // Given a tag string with the format specified in the packa
85     // parseFieldParameters will parse it into a fieldParameters
86     // ignoring unknown parts of the string.
87     func parseFieldParameters(str string) (ret fieldParameters)
88         for _, part := range strings.Split(str, ",") {
89             switch {
90             case part == "optional":
91                 ret.optional = true

```

```

92         case part == "explicit":
93             ret.explicit = true
94             if ret.tag == nil {
95                 ret.tag = new(int)
96             }
97         case part == "ia5":
98             ret.stringType = tagIA5String
99         case part == "printable":
100            ret.stringType = tagPrintableString
101         case strings.HasPrefix(part, "default:"):
102             i, err := strconv.ParseInt(part[8:],
103             if err == nil {
104                 ret.defaultValue = new(int64)
105                 *ret.defaultValue = i
106             }
107         case strings.HasPrefix(part, "tag:"):
108             i, err := strconv.Atoi(part[4:])
109             if err == nil {
110                 ret.tag = new(int)
111                 *ret.tag = i
112             }
113         case part == "set":
114             ret.set = true
115         case part == "application":
116             ret.application = true
117             if ret.tag == nil {
118                 ret.tag = new(int)
119             }
120         case part == "omitempty":
121             ret.omitEmpty = true
122     }
123     }
124     return
125 }
126
127 // Given a reflected Go type, getUniversalType returns the d
128 // and expected compound flag.
129 func getUniversalType(t reflect.Type) (tagNumber int, isComp
130     switch t {
131     case objectIdentifierType:
132         return tagOID, false, true
133     case bitStringType:
134         return tagBitString, false, true
135     case timeType:
136         return tagUTCTime, false, true
137     case enumeratedType:
138         return tagEnum, false, true
139     case bigIntType:
140         return tagInteger, false, true

```

```
141     }
142     switch t.Kind() {
143     case reflect.Bool:
144         return tagBoolean, false, true
145     case reflect.Int, reflect.Int8, reflect.Int16, refle
146         return tagInteger, false, true
147     case reflect.Struct:
148         return tagSequence, true, true
149     case reflect.Slice:
150         if t.Elem().Kind() == reflect.Uint8 {
151             return tagOctetString, false, true
152         }
153         if strings.HasSuffix(t.Name(), "SET") {
154             return tagSet, true, true
155         }
156         return tagSequence, true, true
157     case reflect.String:
158         return tagPrintableString, false, true
159     }
160     return 0, false, false
161 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/asn1/marshal.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package asn1
6
7 import (
8     "bytes"
9     "fmt"
10    "io"
11    "math/big"
12    "reflect"
13    "time"
14 )
15
16 // A forkableWriter is an in-memory buffer that can be
17 // 'forked' to create new forkableWriters that bracket the
18 // original. After
19 //     pre, post := w.fork();
20 // the overall sequence of bytes represented is logically w+
21 type forkableWriter struct {
22     *bytes.Buffer
23     pre, post *forkableWriter
24 }
25
26 func newForkableWriter() *forkableWriter {
27     return &forkableWriter{new(bytes.Buffer), nil, nil}
28 }
29
30 func (f *forkableWriter) fork() (pre, post *forkableWriter) {
31     if f.pre != nil || f.post != nil {
32         panic("have already forked")
33     }
34     f.pre = newForkableWriter()
35     f.post = newForkableWriter()
36     return f.pre, f.post
37 }
38
39 func (f *forkableWriter) Len() (l int) {
40     l += f.Buffer.Len()
41     if f.pre != nil {
```

```

42         l += f.pre.Len()
43     }
44     if f.post != nil {
45         l += f.post.Len()
46     }
47     return
48 }
49
50 func (f *forkableWriter) writeTo(out io.Writer) (n int, err
51     n, err = out.Write(f.Bytes())
52     if err != nil {
53         return
54     }
55
56     var nn int
57
58     if f.pre != nil {
59         nn, err = f.pre.writeTo(out)
60         n += nn
61         if err != nil {
62             return
63         }
64     }
65
66     if f.post != nil {
67         nn, err = f.post.writeTo(out)
68         n += nn
69     }
70     return
71 }
72
73 func marshalBase128Int(out *forkableWriter, n int64) (err er
74     if n == 0 {
75         err = out.WriteByte(0)
76         return
77     }
78
79     l := 0
80     for i := n; i > 0; i >>= 7 {
81         l++
82     }
83
84     for i := l - 1; i >= 0; i-- {
85         o := byte(n >> uint(i*7))
86         o &= 0x7f
87         if i != 0 {
88             o |= 0x80
89         }
90         err = out.WriteByte(o)
91         if err != nil {

```

```

92             return
93         }
94     }
95
96     return nil
97 }
98
99 func marshalInt64(out *forkableWriter, i int64) (err error)
100     n := int64Length(i)
101
102     for ; n > 0; n-- {
103         err = out.WriteByte(byte(i >> uint((n-1)*8)))
104         if err != nil {
105             return
106         }
107     }
108
109     return nil
110 }
111
112 func int64Length(i int64) (numBytes int) {
113     numBytes = 1
114
115     for i > 127 {
116         numBytes++
117         i >>= 8
118     }
119
120     for i < -128 {
121         numBytes++
122         i >>= 8
123     }
124
125     return
126 }
127
128 func marshalBigInt(out *forkableWriter, n *big.Int) (err error)
129     if n.Sign() < 0 {
130         // A negative number has to be converted to
131         // form. So we'll subtract 1 and invert. If
132         // most-significant-bit isn't set then we'll
133         // begin with 0xff in order to keep the
134         nMinus1 := new(big.Int).Neg(n)
135         nMinus1.Sub(nMinus1, bigOne)
136         bytes := nMinus1.Bytes()
137         for i := range bytes {
138             bytes[i] ^= 0xff
139         }
140         if len(bytes) == 0 || bytes[0]&0x80 == 0 {

```

```

141             err = out.WriteByte(0xff)
142             if err != nil {
143                 return
144             }
145         }
146         _, err = out.Write(bytes)
147     } else if n.Sign() == 0 {
148         // Zero is written as a single 0 zero rather
149         err = out.WriteByte(0x00)
150     } else {
151         bytes := n.Bytes()
152         if len(bytes) > 0 && bytes[0]&0x80 != 0 {
153             // We'll have to pad this with 0x00
154             // looking like a negative number.
155             err = out.WriteByte(0)
156             if err != nil {
157                 return
158             }
159         }
160         _, err = out.Write(bytes)
161     }
162     return
163 }
164
165 func marshalLength(out *forkableWriter, i int) (err error) {
166     n := lengthLength(i)
167
168     for ; n > 0; n-- {
169         err = out.WriteByte(byte(i >> uint((n-1)*8)))
170         if err != nil {
171             return
172         }
173     }
174
175     return nil
176 }
177
178 func lengthLength(i int) (numBytes int) {
179     numBytes = 1
180     for i > 255 {
181         numBytes++
182         i >>= 8
183     }
184     return
185 }
186
187 func marshalTagAndLength(out *forkableWriter, t tagAndLength
188     b := uint8(t.class) << 6
189     if t.isCompound {

```

```

190         b |= 0x20
191     }
192     if t.tag >= 31 {
193         b |= 0x1f
194         err = out.WriteByte(b)
195         if err != nil {
196             return
197         }
198         err = marshalBase128Int(out, int64(t.tag))
199         if err != nil {
200             return
201         }
202     } else {
203         b |= uint8(t.tag)
204         err = out.WriteByte(b)
205         if err != nil {
206             return
207         }
208     }
209
210     if t.length >= 128 {
211         l := lengthLength(t.length)
212         err = out.WriteByte(0x80 | byte(l))
213         if err != nil {
214             return
215         }
216         err = marshalLength(out, t.length)
217         if err != nil {
218             return
219         }
220     } else {
221         err = out.WriteByte(byte(t.length))
222         if err != nil {
223             return
224         }
225     }
226
227     return nil
228 }
229
230 func marshalBitString(out *forkableWriter, b BitString) (err
231     paddingBits := byte((8 - b.BitLength%8) % 8)
232     err = out.WriteByte(paddingBits)
233     if err != nil {
234         return
235     }
236     _, err = out.Write(b.Bytes)
237     return
238 }
239

```

```

240 func marshalObjectIdentifier(out *forkableWriter, oid []int)
241     if len(oid) < 2 || oid[0] > 6 || oid[1] >= 40 {
242         return StructuralError{"invalid object ident
243     }
244
245     err = out.WriteByte(byte(oid[0]*40 + oid[1]))
246     if err != nil {
247         return
248     }
249     for i := 2; i < len(oid); i++ {
250         err = marshalBase128Int(out, int64(oid[i]))
251         if err != nil {
252             return
253         }
254     }
255
256     return
257 }
258
259 func marshalPrintableString(out *forkableWriter, s string) (
260     b := []byte(s)
261     for _, c := range b {
262         if !isPrintable(c) {
263             return StructuralError{"PrintableStr
264         }
265     }
266
267     _, err = out.Write(b)
268     return
269 }
270
271 func marshalIA5String(out *forkableWriter, s string) (err er
272     b := []byte(s)
273     for _, c := range b {
274         if c > 127 {
275             return StructuralError{"IA5String co
276         }
277     }
278
279     _, err = out.Write(b)
280     return
281 }
282
283 func marshalTwoDigits(out *forkableWriter, v int) (err error
284     err = out.WriteByte(byte('0' + (v/10)%10))
285     if err != nil {
286         return
287     }
288     return out.WriteByte(byte('0' + v%10))

```

```

289 }
290
291 func marshalUTCtime(out *forkableWriter, t time.Time) (err e
292     utc := t.UTC()
293     year, month, day := utc.Date()
294
295     switch {
296     case 1950 <= year && year < 2000:
297         err = marshalTwoDigits(out, int(year-1900))
298     case 2000 <= year && year < 2050:
299         err = marshalTwoDigits(out, int(year-2000))
300     default:
301         return StructuralError{"Cannot represent tim
302     }
303     if err != nil {
304         return
305     }
306
307     err = marshalTwoDigits(out, int(month))
308     if err != nil {
309         return
310     }
311
312     err = marshalTwoDigits(out, day)
313     if err != nil {
314         return
315     }
316
317     hour, min, sec := utc.Clock()
318
319     err = marshalTwoDigits(out, hour)
320     if err != nil {
321         return
322     }
323
324     err = marshalTwoDigits(out, min)
325     if err != nil {
326         return
327     }
328
329     err = marshalTwoDigits(out, sec)
330     if err != nil {
331         return
332     }
333
334     _, offset := t.Zone()
335
336     switch {
337     case offset/60 == 0:

```

```

338         err = out.WriteByte('Z')
339         return
340     case offset > 0:
341         err = out.WriteByte('+')
342     case offset < 0:
343         err = out.WriteByte('-')
344     }
345
346     if err != nil {
347         return
348     }
349
350     offsetMinutes := offset / 60
351     if offsetMinutes < 0 {
352         offsetMinutes = -offsetMinutes
353     }
354
355     err = marshalTwoDigits(out, offsetMinutes/60)
356     if err != nil {
357         return
358     }
359
360     err = marshalTwoDigits(out, offsetMinutes%60)
361     return
362 }
363
364 func stripTagAndLength(in []byte) []byte {
365     _, offset, err := parseTagAndLength(in, 0)
366     if err != nil {
367         return in
368     }
369     return in[offset:]
370 }
371
372 func marshalBody(out *forkableWriter, value reflect.Value, p
373     switch value.Type() {
374     case timeType:
375         return marshalUTCTime(out, value.Interface())
376     case bitStringType:
377         return marshalBitString(out, value.Interface)
378     case objectIdentifierType:
379         return marshalObjectIdentifier(out, value.In
380     case bigIntType:
381         return marshalBigInt(out, value.Interface()).
382     }
383
384     switch v := value; v.Kind() {
385     case reflect.Bool:
386         if v.Bool() {
387             return out.WriteByte(255)

```

```

388         } else {
389             return out.WriteByte(0)
390         }
391     case reflect.Int, reflect.Int8, reflect.Int16, refle
392         return marshalInt64(out, int64(v.Int()))
393     case reflect.Struct:
394         t := v.Type()
395
396         startingField := 0
397
398         // If the first element of the structure is
399         // RawContents, then we don't bother seriali
400         if t.NumField() > 0 && t.Field(0).Type == ra
401             s := v.Field(0)
402             if s.Len() > 0 {
403                 bytes := make([]byte, s.Len(
404                     for i := 0; i < s.Len(); i++
405                         bytes[i] = uint8(s.I
406                 }
407                 /* The RawContents will cont
408                  * length fields but we'll a
409                  * those ourselves, so we st
410                  * bytes */
411                 _, err = out.Write(stripTagA
412                 return
413             } else {
414                 startingField = 1
415             }
416         }
417
418         for i := startingField; i < t.NumField(); i+
419             var pre *forkableWriter
420             pre, out = out.fork()
421             err = marshalField(pre, v.Field(i),
422             if err != nil {
423                 return
424             }
425         }
426         return
427     case reflect.Slice:
428         sliceType := v.Type()
429         if sliceType.Elem().Kind() == reflect.Uint8
430             bytes := make([]byte, v.Len())
431             for i := 0; i < v.Len(); i++ {
432                 bytes[i] = uint8(v.Index(i).
433             }
434             _, err = out.Write(bytes)
435             return
436     }

```

```

437
438         var params fieldParameters
439         for i := 0; i < v.Len(); i++ {
440             var pre *forkableWriter
441             pre, out = out.fork()
442             err = marshalField(pre, v.Index(i),
443                 if err != nil {
444                     return
445                 }
446         }
447         return
448     case reflect.String:
449         if params.stringType == tagIA5String {
450             return marshalIA5String(out, v.String)
451         } else {
452             return marshalPrintableString(out, v.String)
453         }
454         return
455     }
456
457     return StructuralError{"unknown Go type"}
458 }
459
460 func marshalField(out *forkableWriter, v reflect.Value, para
461 // If the field is an interface{} then recurse into
462 if v.Kind() == reflect.Interface && v.Type().NumMeth
463     return marshalField(out, v.Elem(), params)
464 }
465
466 if v.Kind() == reflect.Slice && v.Len() == 0 && para
467     return
468 }
469
470 if params.optional && reflect.DeepEqual(v.Interface(
471     return
472 }
473
474 if v.Type() == rawValueType {
475     rv := v.Interface().(RawValue)
476     if len(rv.FullBytes) != 0 {
477         _, err = out.Write(rv.FullBytes)
478     } else {
479         err = marshalTagAndLength(out, tagAn
480         if err != nil {
481             return
482         }
483         _, err = out.Write(rv.Bytes)
484     }
485     return

```

```

486     }
487
488     tag, isCompound, ok := getUniversalType(v.Type())
489     if !ok {
490         err = StructuralError{fmt.Sprintf("unknown G
491         return
492     }
493     class := classUniversal
494
495     if params.stringType != 0 {
496         if tag != tagPrintableString {
497             return StructuralError{"Explicit str
498         }
499         tag = params.stringType
500     }
501
502     if params.set {
503         if tag != tagSequence {
504             return StructuralError{"Non sequence
505         }
506         tag = tagSet
507     }
508
509     tags, body := out.fork()
510
511     err = marshalBody(body, v, params)
512     if err != nil {
513         return
514     }
515
516     bodyLen := body.Len()
517
518     var explicitTag *forkableWriter
519     if params.explicit {
520         explicitTag, tags = tags.fork()
521     }
522
523     if !params.explicit && params.tag != nil {
524         // implicit tag.
525         tag = *params.tag
526         class = classContextSpecific
527     }
528
529     err = marshalTagAndLength(tags, tagAndLength{class,
530     if err != nil {
531         return
532     }
533
534     if params.explicit {
535         err = marshalTagAndLength(explicitTag, tagAn

```

```

536             class:      classContextSpecific,
537             tag:        *params.tag,
538             length:     bodyLen + tags.Len(),
539             isCompound: true,
540         })
541     }
542     return nil
543 }
544 }
545
546 // Marshal returns the ASN.1 encoding of val.
547 func Marshal(val interface{}) ([]byte, error) {
548     var out bytes.Buffer
549     v := reflect.ValueOf(val)
550     f := newForkableWriter()
551     err := marshalField(f, v, fieldParameters{})
552     if err != nil {
553         return nil, err
554     }
555     _, err = f.writeTo(&out)
556     return out.Bytes(), nil
557 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/base32/base32.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package base32 implements base32 encoding as specified by
6 package base32
7
8 import (
9     "io"
10    "strconv"
11 )
12
13 /*
14  * Encodings
15  */
16
17 // An Encoding is a radix 32 encoding/decoding scheme, defin
18 // 32-character alphabet. The most common is the "base32" e
19 // introduced for SASL GSSAPI and standardized in RFC 4648.
20 // The alternate "base32hex" encoding is used in DNSSEC.
21 type Encoding struct {
22     encode    string
23     decodeMap [256]byte
24 }
25
26 const encodeStd = "ABCDEFGHIJKLMNOPQRSTUVWXYZ234567"
27 const encodeHex = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
28
29 // NewEncoding returns a new Encoding defined by the given a
30 // which must be a 32-byte string.
31 func NewEncoding(encoder string) *Encoding {
32     e := new(Encoding)
33     e.encode = encoder
34     for i := 0; i < len(e.decodeMap); i++ {
35         e.decodeMap[i] = 0xFF
36     }
37     for i := 0; i < len(encoder); i++ {
38         e.decodeMap[encoder[i]] = byte(i)
39     }
40     return e
41 }
```

```

42
43 // StdEncoding is the standard base32 encoding, as defined i
44 // RFC 4648.
45 var StdEncoding = NewEncoding(encodeStd)
46
47 // HexEncoding is the ``Extended Hex Alphabet'' defined in R
48 // It is typically used in DNS.
49 var HexEncoding = NewEncoding(encodeHex)
50
51 /*
52  * Encoder
53  */
54
55 // Encode encodes src using the encoding enc, writing
56 // EncodedLen(len(src)) bytes to dst.
57 //
58 // The encoding pads the output to a multiple of 8 bytes,
59 // so Encode is not appropriate for use on individual blocks
60 // of a large data stream. Use NewEncoder() instead.
61 func (enc *Encoding) Encode(dst, src []byte) {
62     if len(src) == 0 {
63         return
64     }
65
66     for len(src) > 0 {
67         dst[0] = 0
68         dst[1] = 0
69         dst[2] = 0
70         dst[3] = 0
71         dst[4] = 0
72         dst[5] = 0
73         dst[6] = 0
74         dst[7] = 0
75
76         // Unpack 8x 5-bit source blocks into a 5 by
77         // destination quantum
78         switch len(src) {
79             default:
80                 dst[7] |= src[4] & 0x1F
81                 dst[6] |= src[4] >> 5
82                 fallthrough
83             case 4:
84                 dst[6] |= (src[3] << 3) & 0x1F
85                 dst[5] |= (src[3] >> 2) & 0x1F
86                 dst[4] |= src[3] >> 7
87                 fallthrough
88             case 3:
89                 dst[4] |= (src[2] << 1) & 0x1F
90                 dst[3] |= (src[2] >> 4) & 0x1F
91                 fallthrough

```

```

92         case 2:
93             dst[3] |= (src[1] << 4) & 0x1F
94             dst[2] |= (src[1] >> 1) & 0x1F
95             dst[1] |= (src[1] >> 6) & 0x1F
96             fallthrough
97         case 1:
98             dst[1] |= (src[0] << 2) & 0x1F
99             dst[0] |= src[0] >> 3
100     }
101
102     // Encode 5-bit blocks using the base32 alph
103     for j := 0; j < 8; j++ {
104         dst[j] = enc.encode[dst[j]]
105     }
106
107     // Pad the final quantum
108     if len(src) < 5 {
109         dst[7] = '='
110         if len(src) < 4 {
111             dst[6] = '='
112             dst[5] = '='
113             if len(src) < 3 {
114                 dst[4] = '='
115                 if len(src) < 2 {
116                     dst[3] = '='
117                     dst[2] = '='
118                 }
119             }
120         }
121         break
122     }
123     src = src[5:]
124     dst = dst[8:]
125 }
126 }
127
128 // EncodeToString returns the base32 encoding of src.
129 func (enc *Encoding) EncodeToString(src []byte) string {
130     buf := make([]byte, enc.EncodedLen(len(src)))
131     enc.Encode(buf, src)
132     return string(buf)
133 }
134
135 type encoder struct {
136     err error
137     enc *Encoding
138     w   io.Writer
139     buf [5]byte // buffered data waiting to be encoded
140     nbuf int      // number of bytes in buf

```

```

141         out [1024]byte // output buffer
142     }
143
144     func (e *encoder) Write(p []byte) (n int, err error) {
145         if e.err != nil {
146             return 0, e.err
147         }
148
149         // Leading fringe.
150         if e.nbuf > 0 {
151             var i int
152             for i = 0; i < len(p) && e.nbuf < 5; i++ {
153                 e.buf[e.nbuf] = p[i]
154                 e.nbuf++
155             }
156             n += i
157             p = p[i:]
158             if e.nbuf < 5 {
159                 return
160             }
161             e.enc.Encode(e.out[0:], e.buf[0:])
162             if _, e.err = e.w.Write(e.out[0:8]); e.err !=
163                 return n, e.err
164         }
165         e.nbuf = 0
166     }
167
168     // Large interior chunks.
169     for len(p) >= 5 {
170         nn := len(e.out) / 8 * 5
171         if nn > len(p) {
172             nn = len(p)
173         }
174         nn -= nn % 5
175         if nn > 0 {
176             e.enc.Encode(e.out[0:], p[0:nn])
177             if _, e.err = e.w.Write(e.out[0 : nn]); e.err !=
178                 return n, e.err
179         }
180     }
181     n += nn
182     p = p[nn:]
183 }
184
185 // Trailing fringe.
186 for i := 0; i < len(p); i++ {
187     e.buf[i] = p[i]
188 }
189 e.nbuf = len(p)

```

```

190         n += len(p)
191         return
192     }
193
194     // Close flushes any pending output from the encoder.
195     // It is an error to call Write after calling Close.
196     func (e *encoder) Close() error {
197         // If there's anything left in the buffer, flush it
198         if e.err == nil && e.nbuf > 0 {
199             e.enc.Encode(e.out[0:], e.buf[0:e.nbuf])
200             e.nbuf = 0
201             _, e.err = e.w.Write(e.out[0:8])
202         }
203         return e.err
204     }
205
206     // NewEncoder returns a new base32 stream encoder. Data writ
207     // the returned writer will be encoded using enc and then wr
208     // Base32 encodings operate in 5-byte blocks; when finished
209     // writing, the caller must Close the returned encoder to fl
210     // partially written blocks.
211     func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser {
212         return &encoder{enc: enc, w: w}
213     }
214
215     // EncodedLen returns the length in bytes of the base32 enco
216     // of an input buffer of length n.
217     func (enc *Encoding) EncodedLen(n int) int { return (n + 4)
218
219     /*
220     * Decoder
221     */
222
223     type CorruptInputError int64
224
225     func (e CorruptInputError) Error() string {
226         return "illegal base32 data at input byte " + strcon
227     }
228
229     // decode is like Decode but returns an additional 'end' val
230     // indicates if end-of-message padding was encountered and t
231     // additional data is an error.
232     func (enc *Encoding) decode(dst, src []byte) (n int, end boo
233         osrc := src
234         for len(src) > 0 && !end {
235             // Decode quantum using the base32 alphabet
236             var dbuf [8]byte
237             dlen := 8
238
239             // do the top bytes contain any data?

```

```

240     dbufloop:
241         for j := 0; j < 8; {
242             if len(src) == 0 {
243                 return n, false, CorruptInput
244             }
245             in := src[0]
246             src = src[1:]
247             if in == '\r' || in == '\n' {
248                 // Ignore this character.
249                 continue
250             }
251             if in == '=' && j >= 2 && len(src) <
252                 // We've reached the end and
253                 // padding, the rest should
254                 for k := 0; k < 8-j-1; k++ {
255                     if len(src) > k && s
256                         return n, fa
257                 }
258             }
259             dlen = j
260             end = true
261             break dbufloop
262         }
263         dbuf[j] = enc.decodeMap[in]
264         if dbuf[j] == 0xFF {
265             return n, false, CorruptInput
266         }
267         j++
268     }
269
270     // Pack 8x 5-bit source blocks into 5 byte d
271     // quantum
272     switch dlen {
273     case 7, 8:
274         dst[4] = dbuf[6]<<5 | dbuf[7]
275         fallthrough
276     case 6, 5:
277         dst[3] = dbuf[4]<<7 | dbuf[5]<<2 | d
278         fallthrough
279     case 4:
280         dst[2] = dbuf[3]<<4 | dbuf[4]>>1
281         fallthrough
282     case 3:
283         dst[1] = dbuf[1]<<6 | dbuf[2]<<1 | d
284         fallthrough
285     case 2:
286         dst[0] = dbuf[0]<<3 | dbuf[1]>>2
287     }
288     dst = dst[5:]

```

```

289         switch dlen {
290             case 2:
291                 n += 1
292             case 3, 4:
293                 n += 2
294             case 5:
295                 n += 3
296             case 6, 7:
297                 n += 4
298             case 8:
299                 n += 5
300         }
301     }
302     return n, end, nil
303 }
304
305 // Decode decodes src using the encoding enc. It writes at
306 // DecodedLen(len(src)) bytes to dst and returns the number
307 // written. If src contains invalid base32 data, it will re
308 // number of bytes successfully written and CorruptInputErro
309 // New line characters (\r and \n) are ignored.
310 func (enc *Encoding) Decode(dst, src []byte) (n int, err error) {
311     n, _, err = enc.decode(dst, src)
312     return
313 }
314
315 // DecodeString returns the bytes represented by the base32
316 func (enc *Encoding) DecodeString(s string) ([]byte, error) {
317     dbuf := make([]byte, enc.DecodedLen(len(s)))
318     n, err := enc.Decode(dbuf, []byte(s))
319     return dbuf[:n], err
320 }
321
322 type decoder struct {
323     err    error
324     enc    *Encoding
325     r      io.Reader
326     end    bool // saw end of message
327     buf    [1024]byte // leftover input
328     nbuf   int
329     out    []byte // leftover decoded output
330     outbuf [1024 / 8 * 5]byte
331 }
332
333 func (d *decoder) Read(p []byte) (n int, err error) {
334     if d.err != nil {
335         return 0, d.err
336     }
337

```

```

338         // Use leftover decoded output from last read.
339         if len(d.out) > 0 {
340             n = copy(p, d.out)
341             d.out = d.out[n:]
342             return n, nil
343         }
344
345         // Read a chunk.
346         nn := len(p) / 5 * 8
347         if nn < 8 {
348             nn = 8
349         }
350         if nn > len(d.buf) {
351             nn = len(d.buf)
352         }
353         nn, d.err = io.ReadAtLeast(d.r, d.buf[d.nbuf:nn], 8-
354             d.nbuf += nn
355         if d.nbuf < 8 {
356             return 0, d.err
357         }
358
359         // Decode chunk into p, or d.out and then p if p is
360         nr := d.nbuf / 8 * 8
361         nw := d.nbuf / 8 * 5
362         if nw > len(p) {
363             nw, d.end, d.err = d.enc.decode(d.outbuf[0:])
364             d.out = d.outbuf[0:nw]
365             n = copy(p, d.out)
366             d.out = d.out[n:]
367         } else {
368             n, d.end, d.err = d.enc.decode(p, d.buf[0:nr]
369         }
370         d.nbuf -= nr
371         for i := 0; i < d.nbuf; i++ {
372             d.buf[i] = d.buf[i+nr]
373         }
374
375         if d.err == nil {
376             d.err = err
377         }
378         return n, d.err
379     }
380
381     // NewDecoder constructs a new base32 stream decoder.
382     func NewDecoder(enc *Encoding, r io.Reader) io.Reader {
383         return &decoder{enc: enc, r: r}
384     }
385
386     // DecodedLen returns the maximum length in bytes of the dec
387     // corresponding to n bytes of base32-encoded data.

```

```
388 func (enc *Encoding) DecodedLen(n int) int { return n / 8 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/base64/base64.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package base64 implements base64 encoding as specified by
6 package base64
7
8 import (
9     "io"
10    "strconv"
11 )
12
13 /*
14  * Encodings
15  */
16
17 // An Encoding is a radix 64 encoding/decoding scheme, defin
18 // 64-character alphabet. The most common encoding is the "
19 // encoding defined in RFC 4648 and used in MIME (RFC 2045)
20 // (RFC 1421). RFC 4648 also defines an alternate encoding,
21 // the standard encoding with - and _ substituted for + and
22 type Encoding struct {
23     encode      string
24     decodeMap [256]byte
25 }
26
27 const encodeStd = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
28 const encodeURL = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
29
30 // NewEncoding returns a new Encoding defined by the given a
31 // which must be a 64-byte string.
32 func NewEncoding(encoder string) *Encoding {
33     e := new(Encoding)
34     e.encode = encoder
35     for i := 0; i < len(e.decodeMap); i++ {
36         e.decodeMap[i] = 0xFF
37     }
38     for i := 0; i < len(encoder); i++ {
39         e.decodeMap[encoder[i]] = byte(i)
40     }
41     return e

```

```

42 }
43
44 // StdEncoding is the standard base64 encoding, as defined i
45 // RFC 4648.
46 var StdEncoding = NewEncoding(encodeStd)
47
48 // URLEncoding is the alternate base64 encoding defined in R
49 // It is typically used in URLs and file names.
50 var URLEncoding = NewEncoding(encodeURL)
51
52 /*
53  * Encoder
54  */
55
56 // Encode encodes src using the encoding enc, writing
57 // EncodedLen(len(src)) bytes to dst.
58 //
59 // The encoding pads the output to a multiple of 4 bytes,
60 // so Encode is not appropriate for use on individual blocks
61 // of a large data stream. Use NewEncoder() instead.
62 func (enc *Encoding) Encode(dst, src []byte) {
63     if len(src) == 0 {
64         return
65     }
66
67     for len(src) > 0 {
68         dst[0] = 0
69         dst[1] = 0
70         dst[2] = 0
71         dst[3] = 0
72
73         // Unpack 4x 6-bit source blocks into a 4 by
74         // destination quantum
75         switch len(src) {
76         default:
77             dst[3] |= src[2] & 0x3F
78             dst[2] |= src[2] >> 6
79             fallthrough
80         case 2:
81             dst[2] |= (src[1] << 2) & 0x3F
82             dst[1] |= src[1] >> 4
83             fallthrough
84         case 1:
85             dst[1] |= (src[0] << 4) & 0x3F
86             dst[0] |= src[0] >> 2
87         }
88
89         // Encode 6-bit blocks using the base64 alph
90         for j := 0; j < 4; j++ {
91             dst[j] = enc.encode[dst[j]]

```

```

92         }
93
94         // Pad the final quantum
95         if len(src) < 3 {
96             dst[3] = '='
97             if len(src) < 2 {
98                 dst[2] = '='
99             }
100            break
101        }
102
103        src = src[3:]
104        dst = dst[4:]
105    }
106 }
107
108 // EncodeToString returns the base64 encoding of src.
109 func (enc *Encoding) EncodeToString(src []byte) string {
110     buf := make([]byte, enc.EncodedLen(len(src)))
111     enc.Encode(buf, src)
112     return string(buf)
113 }
114
115 type encoder struct {
116     err error
117     enc *Encoding
118     w   io.Writer
119     buf [3]byte // buffered data waiting to be encoded
120     nbuf int    // number of bytes in buf
121     out [1024]byte // output buffer
122 }
123
124 func (e *encoder) Write(p []byte) (n int, err error) {
125     if e.err != nil {
126         return 0, e.err
127     }
128
129     // Leading fringe.
130     if e.nbuf > 0 {
131         var i int
132         for i = 0; i < len(p) && e.nbuf < 3; i++ {
133             e.buf[e.nbuf] = p[i]
134             e.nbuf++
135         }
136         n += i
137         p = p[i:]
138         if e.nbuf < 3 {
139             return
140         }

```

```

141         e.enc.Encode(e.out[0:], e.buf[0:])
142         if _, e.err = e.w.Write(e.out[0:4]); e.err !=
143             return n, e.err
144     }
145     e.nbuf = 0
146 }
147
148 // Large interior chunks.
149 for len(p) >= 3 {
150     nn := len(e.out) / 4 * 3
151     if nn > len(p) {
152         nn = len(p)
153     }
154     nn -= nn % 3
155     if nn > 0 {
156         e.enc.Encode(e.out[0:], p[0:nn])
157         if _, e.err = e.w.Write(e.out[0 : nn]); e.err !=
158             return n, e.err
159     }
160 }
161     n += nn
162     p = p[nn:]
163 }
164
165 // Trailing fringe.
166 for i := 0; i < len(p); i++ {
167     e.buf[i] = p[i]
168 }
169 e.nbuf = len(p)
170 n += len(p)
171 return
172 }
173
174 // Close flushes any pending output from the encoder.
175 // It is an error to call Write after calling Close.
176 func (e *encoder) Close() error {
177     // If there's anything left in the buffer, flush it
178     if e.err == nil && e.nbuf > 0 {
179         e.enc.Encode(e.out[0:], e.buf[0:e.nbuf])
180         e.nbuf = 0
181         _, e.err = e.w.Write(e.out[0:4])
182     }
183     return e.err
184 }
185
186 // NewEncoder returns a new base64 stream encoder. Data writ
187 // the returned writer will be encoded using enc and then wr
188 // Base64 encodings operate in 4-byte blocks; when finished
189 // writing, the caller must Close the returned encoder to fl

```

```

190 // partially written blocks.
191 func NewEncoder(enc *Encoding, w io.Writer) io.WriteCloser {
192     return &encoder{enc: enc, w: w}
193 }
194
195 // EncodedLen returns the length in bytes of the base64 enco
196 // of an input buffer of length n.
197 func (enc *Encoding) EncodedLen(n int) int { return (n + 2)
198
199 /*
200  * Decoder
201  */
202
203 type CorruptInputError int64
204
205 func (e CorruptInputError) Error() string {
206     return "illegal base64 data at input byte " + strcon
207 }
208
209 // decode is like Decode but returns an additional 'end' val
210 // indicates if end-of-message padding was encountered and t
211 // additional data is an error.
212 func (enc *Encoding) decode(dst, src []byte) (n int, end boo
213     osrc := src
214     for len(src) > 0 && !end {
215         // Decode quantum using the base64 alphabet
216         var dbuf [4]byte
217         dlen := 4
218
219         dbufloop:
220             for j := 0; j < 4; {
221                 if len(src) == 0 {
222                     return n, false, CorruptInpu
223                 }
224                 in := src[0]
225                 src = src[1:]
226                 if in == '\r' || in == '\n' {
227                     // Ignore this character.
228                     continue
229                 }
230                 if in == '=' && j >= 2 && len(src) <
231                     // We've reached the end and
232                     // padding
233                     if len(src) == 0 && j == 2 {
234                         // not enough paddin
235                         return n, false, Cor
236                     }
237                     if len(src) > 0 && src[0] !=
238                         // incorrect padding
239                         return n, false, Cor

```

```

240         }
241         dlen = j
242         end = true
243         break dbufloop
244     }
245     dbuf[j] = enc.decodeMap[in]
246     if dbuf[j] == 0xFF {
247         return n, false, CorruptInputError
248     }
249     j++
250 }
251
252 // Pack 4x 6-bit source blocks into 3 byte d
253 // quantum
254 switch dlen {
255 case 4:
256     dst[2] = dbuf[2]<<6 | dbuf[3]
257     fallthrough
258 case 3:
259     dst[1] = dbuf[1]<<4 | dbuf[2]>>2
260     fallthrough
261 case 2:
262     dst[0] = dbuf[0]<<2 | dbuf[1]>>4
263 }
264 dst = dst[3:]
265 n += dlen - 1
266 }
267
268 return n, end, nil
269 }
270
271 // Decode decodes src using the encoding enc. It writes at
272 // DecodedLen(len(src)) bytes to dst and returns the number
273 // written. If src contains invalid base64 data, it will re
274 // number of bytes successfully written and CorruptInputError
275 // New line characters (\r and \n) are ignored.
276 func (enc *Encoding) Decode(dst []byte) (n int, err error) {
277     n, _, err = enc.decode(dst, src)
278     return
279 }
280
281 // DecodeString returns the bytes represented by the base64
282 func (enc *Encoding) DecodeString(s string) ([]byte, error) {
283     dbuf := make([]byte, enc.DecodedLen(len(s)))
284     n, err := enc.Decode(dbuf, []byte(s))
285     return dbuf[:n], err
286 }
287
288 type decoder struct {

```

```

289     err    error
290     enc    *Encoding
291     r      io.Reader
292     end    bool        // saw end of message
293     buf    [1024]byte  // leftover input
294     nbuf   int
295     out    []byte     // leftover decoded output
296     outbuf [1024 / 4 * 3]byte
297 }
298
299 func (d *decoder) Read(p []byte) (n int, err error) {
300     if d.err != nil {
301         return 0, d.err
302     }
303
304     // Use leftover decoded output from last read.
305     if len(d.out) > 0 {
306         n = copy(p, d.out)
307         d.out = d.out[n:]
308         return n, nil
309     }
310
311     // Read a chunk.
312     nn := len(p) / 3 * 4
313     if nn < 4 {
314         nn = 4
315     }
316     if nn > len(d.buf) {
317         nn = len(d.buf)
318     }
319     nn, d.err = io.ReadAtLeast(d.r, d.buf[d.nbuf:nn], 4-
320     d.nbuf += nn
321     if d.nbuf < 4 {
322         return 0, d.err
323     }
324
325     // Decode chunk into p, or d.out and then p if p is
326     nr := d.nbuf / 4 * 4
327     nw := d.nbuf / 4 * 3
328     if nw > len(p) {
329         nw, d.end, d.err = d.enc.decode(d.outbuf[0:])
330         d.out = d.outbuf[0:nw]
331         n = copy(p, d.out)
332         d.out = d.out[n:]
333     } else {
334         n, d.end, d.err = d.enc.decode(p, d.buf[0:nr]
335     }
336     d.nbuf -= nr
337     for i := 0; i < d.nbuf; i++ {

```

```
338             d.buf[i] = d.buf[i+nr]
339         }
340
341         if d.err == nil {
342             d.err = err
343         }
344         return n, d.err
345     }
346
347     // NewDecoder constructs a new base64 stream decoder.
348     func NewDecoder(enc *Encoding, r io.Reader) io.Reader {
349         return &decoder{enc: enc, r: r}
350     }
351
352     // DecodedLen returns the maximum length in bytes of the dec
353     // corresponding to n bytes of base64-encoded data.
354     func (enc *Encoding) DecodedLen(n int) int { return n / 4 *
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/binary/binary.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package binary implements translation between numbers and
6 // and encoding and decoding of varints.
7 //
8 // Numbers are translated by reading and writing fixed-size
9 // A fixed-size value is either a fixed-size arithmetic
10 // type (int8, uint8, int16, float32, complex64, ...)
11 // or an array or struct containing only fixed-size values.
12 //
13 // Varints are a method of encoding integers using one or mo
14 // numbers with smaller absolute value take a smaller number
15 // For a specification, see http://code.google.com/apis/prot
16 package binary
17
18 import (
19     "errors"
20     "io"
21     "math"
22     "reflect"
23 )
24
25 // A ByteOrder specifies how to convert byte sequences into
26 // 16-, 32-, or 64-bit unsigned integers.
27 type ByteOrder interface {
28     Uint16([]byte) uint16
29     Uint32([]byte) uint32
30     Uint64([]byte) uint64
31     PutUint16([]byte, uint16)
32     PutUint32([]byte, uint32)
33     PutUint64([]byte, uint64)
34     String() string
35 }
36
37 // LittleEndian is the little-endian implementation of ByteO
38 var LittleEndian littleEndian
39
40 // BigEndian is the big-endian implementation of ByteOrder.
41 var BigEndian bigEndian
```

```

42
43 type littleEndian struct{}
44
45 func (littleEndian) Uint16(b []byte) uint16 { return uint16(
46
47 func (littleEndian) PutUint16(b []byte, v uint16) {
48     b[0] = byte(v)
49     b[1] = byte(v >> 8)
50 }
51
52 func (littleEndian) Uint32(b []byte) uint32 {
53     return uint32(b[0]) | uint32(b[1])<<8 | uint32(b[2])
54 }
55
56 func (littleEndian) PutUint32(b []byte, v uint32) {
57     b[0] = byte(v)
58     b[1] = byte(v >> 8)
59     b[2] = byte(v >> 16)
60     b[3] = byte(v >> 24)
61 }
62
63 func (littleEndian) Uint64(b []byte) uint64 {
64     return uint64(b[0]) | uint64(b[1])<<8 | uint64(b[2])
65     uint64(b[4])<<32 | uint64(b[5])<<40 | uint64
66 }
67
68 func (littleEndian) PutUint64(b []byte, v uint64) {
69     b[0] = byte(v)
70     b[1] = byte(v >> 8)
71     b[2] = byte(v >> 16)
72     b[3] = byte(v >> 24)
73     b[4] = byte(v >> 32)
74     b[5] = byte(v >> 40)
75     b[6] = byte(v >> 48)
76     b[7] = byte(v >> 56)
77 }
78
79 func (littleEndian) String() string { return "LittleEndian"
80
81 func (littleEndian) GoString() string { return "binary.Littl
82
83 type bigEndian struct{}
84
85 func (bigEndian) Uint16(b []byte) uint16 { return uint16(b[1]
86
87 func (bigEndian) PutUint16(b []byte, v uint16) {
88     b[0] = byte(v >> 8)
89     b[1] = byte(v)
90 }
91

```

```

92 func (bigEndian) Uint32(b []byte) uint32 {
93     return uint32(b[3]) | uint32(b[2])<<8 | uint32(b[1])
94 }
95
96 func (bigEndian) PutUint32(b []byte, v uint32) {
97     b[0] = byte(v >> 24)
98     b[1] = byte(v >> 16)
99     b[2] = byte(v >> 8)
100    b[3] = byte(v)
101 }
102
103 func (bigEndian) Uint64(b []byte) uint64 {
104     return uint64(b[7]) | uint64(b[6])<<8 | uint64(b[5])
105         uint64(b[3])<<32 | uint64(b[2])<<40 | uint64
106 }
107
108 func (bigEndian) PutUint64(b []byte, v uint64) {
109     b[0] = byte(v >> 56)
110     b[1] = byte(v >> 48)
111     b[2] = byte(v >> 40)
112     b[3] = byte(v >> 32)
113     b[4] = byte(v >> 24)
114     b[5] = byte(v >> 16)
115     b[6] = byte(v >> 8)
116     b[7] = byte(v)
117 }
118
119 func (bigEndian) String() string { return "BigEndian" }
120
121 func (bigEndian) GoString() string { return "binary.BigEndi
122
123 // Read reads structured binary data from r into data.
124 // Data must be a pointer to a fixed-size value or a slice
125 // of fixed-size values.
126 // Bytes read from r are decoded using the specified byte or
127 // and written to successive fields of the data.
128 func Read(r io.Reader, order ByteOrder, data interface{}) er
129     // Fast path for basic types.
130     if n := intDestSize(data); n != 0 {
131         var b [8]byte
132         bs := b[:n]
133         if _, err := io.ReadFull(r, bs); err != nil
134             return err
135     }
136     switch v := data.(type) {
137     case *int8:
138         *v = int8(b[0])
139     case *uint8:
140         *v = b[0]

```

```

141         case *int16:
142             *v = int16(order.Uint16(bs))
143         case *uint16:
144             *v = order.Uint16(bs)
145         case *int32:
146             *v = int32(order.Uint32(bs))
147         case *uint32:
148             *v = order.Uint32(bs)
149         case *int64:
150             *v = int64(order.Uint64(bs))
151         case *uint64:
152             *v = order.Uint64(bs)
153     }
154     return nil
155 }
156
157 // Fallback to reflect-based.
158 var v reflect.Value
159 switch d := reflect.ValueOf(data); d.Kind() {
160 case reflect.Ptr:
161     v = d.Elem()
162 case reflect.Slice:
163     v = d
164 default:
165     return errors.New("binary.Read: invalid type")
166 }
167 size := dataSize(v)
168 if size < 0 {
169     return errors.New("binary.Read: invalid type")
170 }
171 d := &decoder{order: order, buf: make([]byte, size)}
172 if _, err := io.ReadFull(r, d.buf); err != nil {
173     return err
174 }
175 d.value(v)
176 return nil
177 }
178
179 // Write writes the binary representation of data into w.
180 // Data must be a fixed-size value or a slice of fixed-size
181 // values, or a pointer to such data.
182 // Bytes written to w are encoded using the specified byte o
183 // and read from successive fields of the data.
184 func Write(w io.Writer, order ByteOrder, data interface{}) e
185     // Fast path for basic types.
186     var b [8]byte
187     var bs []byte
188     switch v := data.(type) {
189     case *int8:

```

```

190         bs = b[:1]
191         b[0] = byte(*v)
192     case int8:
193         bs = b[:1]
194         b[0] = byte(v)
195     case *uint8:
196         bs = b[:1]
197         b[0] = *v
198     case uint8:
199         bs = b[:1]
200         b[0] = byte(v)
201     case *int16:
202         bs = b[:2]
203         order.PutUint16(bs, uint16(*v))
204     case int16:
205         bs = b[:2]
206         order.PutUint16(bs, uint16(v))
207     case *uint16:
208         bs = b[:2]
209         order.PutUint16(bs, *v)
210     case uint16:
211         bs = b[:2]
212         order.PutUint16(bs, v)
213     case *int32:
214         bs = b[:4]
215         order.PutUint32(bs, uint32(*v))
216     case int32:
217         bs = b[:4]
218         order.PutUint32(bs, uint32(v))
219     case *uint32:
220         bs = b[:4]
221         order.PutUint32(bs, *v)
222     case uint32:
223         bs = b[:4]
224         order.PutUint32(bs, v)
225     case *int64:
226         bs = b[:8]
227         order.PutUint64(bs, uint64(*v))
228     case int64:
229         bs = b[:8]
230         order.PutUint64(bs, uint64(v))
231     case *uint64:
232         bs = b[:8]
233         order.PutUint64(bs, *v)
234     case uint64:
235         bs = b[:8]
236         order.PutUint64(bs, v)
237     }
238     if bs != nil {
239         _, err := w.Write(bs)

```

```

240         return err
241     }
242     v := reflect.Indirect(reflect.ValueOf(data))
243     size := dataSize(v)
244     if size < 0 {
245         return errors.New("binary.Write: invalid typ
246     }
247     buf := make([]byte, size)
248     e := &encoder{order: order, buf: buf}
249     e.value(v)
250     _, err := w.Write(buf)
251     return err
252 }
253
254 // Size returns how many bytes Write would generate to encod
255 // must be a fixed-size value or a slice of fixed-size value
256 func Size(v interface{}) int {
257     return dataSize(reflect.Indirect(reflect.ValueOf(v))
258 }
259
260 // dataSize returns the number of bytes the actual data repr
261 // For compound structures, it sums the sizes of the element
262 // it returns the length of the slice times the element size
263 // occupied by the header.
264 func dataSize(v reflect.Value) int {
265     if v.Kind() == reflect.Slice {
266         elem := sizeof(v.Type().Elem())
267         if elem < 0 {
268             return -1
269         }
270         return v.Len() * elem
271     }
272     return sizeof(v.Type())
273 }
274
275 func sizeof(t reflect.Type) int {
276     switch t.Kind() {
277     case reflect.Array:
278         n := sizeof(t.Elem())
279         if n < 0 {
280             return -1
281         }
282         return t.Len() * n
283
284     case reflect.Struct:
285         sum := 0
286         for i, n := 0, t.NumField(); i < n; i++ {
287             s := sizeof(t.Field(i).Type)
288             if s < 0 {

```

```

289             return -1
290         }
291         sum += s
292     }
293     return sum
294
295     case reflect.Uint8, reflect.Uint16, reflect.Uint32,
296         reflect.Int8, reflect.Int16, reflect.Int32,
297         reflect.Float32, reflect.Float64, reflect.Co
298     return int(t.Size())
299 }
300 return -1
301 }
302
303 type decoder struct {
304     order ByteOrder
305     buf []byte
306 }
307
308 type encoder struct {
309     order ByteOrder
310     buf []byte
311 }
312
313 func (d *decoder) uint8() uint8 {
314     x := d.buf[0]
315     d.buf = d.buf[1:]
316     return x
317 }
318
319 func (e *encoder) uint8(x uint8) {
320     e.buf[0] = x
321     e.buf = e.buf[1:]
322 }
323
324 func (d *decoder) uint16() uint16 {
325     x := d.order.Uint16(d.buf[0:2])
326     d.buf = d.buf[2:]
327     return x
328 }
329
330 func (e *encoder) uint16(x uint16) {
331     e.order.PutUint16(e.buf[0:2], x)
332     e.buf = e.buf[2:]
333 }
334
335 func (d *decoder) uint32() uint32 {
336     x := d.order.Uint32(d.buf[0:4])
337     d.buf = d.buf[4:]

```

```

338         return x
339     }
340
341     func (e *encoder) uint32(x uint32) {
342         e.order.PutUint32(e.buf[0:4], x)
343         e.buf = e.buf[4:]
344     }
345
346     func (d *decoder) uint64() uint64 {
347         x := d.order.Uint64(d.buf[0:8])
348         d.buf = d.buf[8:]
349         return x
350     }
351
352     func (e *encoder) uint64(x uint64) {
353         e.order.PutUint64(e.buf[0:8], x)
354         e.buf = e.buf[8:]
355     }
356
357     func (d *decoder) int8() int8 { return int8(d.uint8()) }
358
359     func (e *encoder) int8(x int8) { e.uint8(uint8(x)) }
360
361     func (d *decoder) int16() int16 { return int16(d.uint16()) }
362
363     func (e *encoder) int16(x int16) { e.uint16(uint16(x)) }
364
365     func (d *decoder) int32() int32 { return int32(d.uint32()) }
366
367     func (e *encoder) int32(x int32) { e.uint32(uint32(x)) }
368
369     func (d *decoder) int64() int64 { return int64(d.uint64()) }
370
371     func (e *encoder) int64(x int64) { e.uint64(uint64(x)) }
372
373     func (d *decoder) value(v reflect.Value) {
374         switch v.Kind() {
375             case reflect.Array:
376                 l := v.Len()
377                 for i := 0; i < l; i++ {
378                     d.value(v.Index(i))
379                 }
380
381             case reflect.Struct:
382                 l := v.NumField()
383                 for i := 0; i < l; i++ {
384                     d.value(v.Field(i))
385                 }
386
387             case reflect.Slice:

```

```

388         l := v.Len()
389         for i := 0; i < l; i++ {
390             d.value(v.Index(i))
391         }
392
393     case reflect.Int8:
394         v.SetInt(int64(d.int8()))
395     case reflect.Int16:
396         v.SetInt(int64(d.int16()))
397     case reflect.Int32:
398         v.SetInt(int64(d.int32()))
399     case reflect.Int64:
400         v.SetInt(d.int64())
401
402     case reflect.Uint8:
403         v.SetUint(uint64(d.uint8()))
404     case reflect.Uint16:
405         v.SetUint(uint64(d.uint16()))
406     case reflect.Uint32:
407         v.SetUint(uint64(d.uint32()))
408     case reflect.Uint64:
409         v.SetUint(d.uint64())
410
411     case reflect.Float32:
412         v.SetFloat(float64(math.Float32frombits(d.ui
413     case reflect.Float64:
414         v.SetFloat(math.Float64frombits(d.uint64())))
415
416     case reflect.Complex64:
417         v.SetComplex(complex(
418             float64(math.Float32frombits(d.uint3
419             float64(math.Float32frombits(d.uint3
420         ))
421     case reflect.Complex128:
422         v.SetComplex(complex(
423             math.Float64frombits(d.uint64()),
424             math.Float64frombits(d.uint64()),
425         ))
426     }
427 }
428
429 func (e *encoder) value(v reflect.Value) {
430     switch v.Kind() {
431     case reflect.Array:
432         l := v.Len()
433         for i := 0; i < l; i++ {
434             e.value(v.Index(i))
435         }
436

```

```

437     case reflect.Struct:
438         l := v.NumField()
439         for i := 0; i < l; i++ {
440             e.value(v.Field(i))
441         }
442
443     case reflect.Slice:
444         l := v.Len()
445         for i := 0; i < l; i++ {
446             e.value(v.Index(i))
447         }
448
449     case reflect.Int, reflect.Int8, reflect.Int16, refle
450         switch v.Type().Kind() {
451         case reflect.Int8:
452             e.int8(int8(v.Int()))
453         case reflect.Int16:
454             e.int16(int16(v.Int()))
455         case reflect.Int32:
456             e.int32(int32(v.Int()))
457         case reflect.Int64:
458             e.int64(v.Int())
459         }
460
461     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
462         switch v.Type().Kind() {
463         case reflect.Uint8:
464             e.uint8(uint8(v.Uint()))
465         case reflect.Uint16:
466             e.uint16(uint16(v.Uint()))
467         case reflect.Uint32:
468             e.uint32(uint32(v.Uint()))
469         case reflect.Uint64:
470             e.uint64(v.Uint())
471         }
472
473     case reflect.Float32, reflect.Float64:
474         switch v.Type().Kind() {
475         case reflect.Float32:
476             e.uint32(math.Float32bits(float32(v.
477         case reflect.Float64:
478             e.uint64(math.Float64bits(v.Float()))
479         }
480
481     case reflect.Complex64, reflect.Complex128:
482         switch v.Type().Kind() {
483         case reflect.Complex64:
484             x := v.Complex()
485             e.uint32(math.Float32bits(float32(re

```

```

486             e.uint32(math.Float32bits(float32(ir
487         case reflect.Complex128:
488             x := v.Complex()
489             e.uint64(math.Float64bits(real(x)))
490             e.uint64(math.Float64bits(imag(x)))
491         }
492     }
493 }
494
495 // intDestSize returns the size of the integer that ptrType
496 // or 0 if the type is not supported.
497 func intDestSize(ptrType interface{}) int {
498     switch ptrType.(type) {
499     case *int8, *uint8:
500         return 1
501     case *int16, *uint16:
502         return 2
503     case *int32, *uint32:
504         return 4
505     case *int64, *uint64:
506         return 8
507     }
508     return 0
509 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/binary/varint.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package binary
6
7 // This file implements "varint" encoding of 64-bit integers
8 // The encoding is:
9 // - unsigned integers are serialized 7 bits at a time, starting
10 //   from the least significant bits
11 // - the most significant bit (msb) in each output byte indicates
12 //   whether it is a continuation byte (msb = 1)
13 // - signed integers are mapped to unsigned integers using the two's
14 //   complement encoding: Positive values x are written as 2*x + 0, negative
15 //   values are written as 2*(^x) + 1; that is, negative numbers are
16 //   written as their two's complement and whether to complement is encoded
17 //   in bit 0.
18 // Design note:
19 // At most 10 bytes are needed for 64-bit values. The encoding could
20 // be more dense: a full 64-bit value needs an extra byte just to
21 // know there can't be more than 64 bits. This is a trivial optimization
22 // that would reduce the maximum encoding length to 9 bytes. However, it is
23 // invariant that the msb is always the "continuation bit" and this
24 // format is incompatible with a varint encoding for larger numbers.
25
26
27 import (
28     "errors"
29     "io"
30 )
31
32 // MaxVarintLenN is the maximum length of a varint-encoded N
33 const (
34     MaxVarintLen16 = 3
35     MaxVarintLen32 = 5
36     MaxVarintLen64 = 10
37 )
38
39 // PutUvarint encodes a uint64 into buf and returns the number of bytes
40 // written. If the buffer is too small, PutUvarint will panic.
41 func PutUvarint(buf []byte, x uint64) int {
```

```

42         i := 0
43         for x >= 0x80 {
44             buf[i] = byte(x) | 0x80
45             x >>= 7
46             i++
47         }
48         buf[i] = byte(x)
49         return i + 1
50     }
51
52     // Uvarint decodes a uint64 from buf and returns that value
53     // number of bytes read (> 0). If an error occurred, the val
54     // and the number of bytes n is <= 0 meaning:
55     //
56     //     n == 0: buf too small
57     //     n < 0: value larger than 64 bits (overflow)
58     //             and -n is the number of bytes read
59     //
60     func Uvarint(buf []byte) (uint64, int) {
61         var x uint64
62         var s uint
63         for i, b := range buf {
64             if b < 0x80 {
65                 if i > 9 || i == 9 && b > 1 {
66                     return 0, -(i + 1) // overfl
67                 }
68                 return x | uint64(b)<<s, i + 1
69             }
70             x |= uint64(b&0x7f) << s
71             s += 7
72         }
73         return 0, 0
74     }
75
76     // PutVarint encodes an int64 into buf and returns the numbe
77     // If the buffer is too small, PutVarint will panic.
78     func PutVarint(buf []byte, x int64) int {
79         ux := uint64(x) << 1
80         if x < 0 {
81             ux = ^ux
82         }
83         return PutUvarint(buf, ux)
84     }
85
86     // Varint decodes an int64 from buf and returns that value a
87     // number of bytes read (> 0). If an error occurred, the val
88     // and the number of bytes n is <= 0 with the following mean
89     //
90     //     n == 0: buf too small
91     //     n < 0: value larger than 64 bits (overflow)

```

```

92 //          and -n is the number of bytes read
93 //
94 func Varint(buf []byte) (int64, int) {
95     ux, n := Uvarint(buf) // ok to continue in presence
96     x := int64(ux >> 1)
97     if ux&1 != 0 {
98         x = ^x
99     }
100    return x, n
101 }
102
103 var overflow = errors.New("binary: varint overflows a 64-bit
104
105 // ReadUvarint reads an encoded unsigned integer from r and
106 func ReadUvarint(r io.ByteReader) (uint64, error) {
107     var x uint64
108     var s uint
109     for i := 0; ; i++ {
110         b, err := r.ReadByte()
111         if err != nil {
112             return x, err
113         }
114         if b < 0x80 {
115             if i > 9 || i == 9 && b > 1 {
116                 return x, overflow
117             }
118             return x | uint64(b)<<s, nil
119         }
120         x |= uint64(b&0x7f) << s
121         s += 7
122     }
123     panic("unreachable")
124 }
125
126 // ReadVarint reads an encoded unsigned integer from r and r
127 func ReadVarint(r io.ByteReader) (int64, error) {
128     ux, err := ReadUvarint(r) // ok to continue in prese
129     x := int64(ux >> 1)
130     if ux&1 != 0 {
131         x = ^x
132     }
133     return x, err
134 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/csv/reader.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package csv reads and writes comma-separated values (CSV)
6 //
7 // A csv file contains zero or more records of one or more f
8 // Each record is separated by the newline character. The fi
9 // optionally be followed by a newline character.
10 //
11 //      field1,field2,field3
12 //
13 // White space is considered part of a field.
14 //
15 // Carriage returns before newline characters are silently r
16 //
17 // Blank lines are ignored. A line with only whitespace cha
18 // the ending newline character) is not considered a blank l
19 //
20 // Fields which start and stop with the quote character " ar
21 // quoted-fields. The beginning and ending quote are not pa
22 // field.
23 //
24 // The source:
25 //
26 //      normal string,"quoted-field"
27 //
28 // results in the fields
29 //
30 //      {`normal string`, `quoted-field`}
31 //
32 // Within a quoted-field a quote character followed by a sec
33 // character is considered a single quote.
34 //
35 //      "the ""word"" is true","a ""quoted-field""
36 //
37 // results in
38 //
39 //      {`the "word" is true`, `a "quoted-field"`}
40 //
41 // Newlines and commas may be included in a quoted-field
```

```

42 //
43 //     "Multi-line
44 //     field","comma is ,"
45 //
46 // results in
47 //
48 //     {`Multi-line
49 //     field`, `comma is ,`}
50 package csv
51
52 import (
53     "bufio"
54     "bytes"
55     "errors"
56     "fmt"
57     "io"
58     "unicode"
59 )
60
61 // A ParseError is returned for parsing errors.
62 // The first line is 1. The first column is 0.
63 type ParseError struct {
64     Line    int    // Line where the error occurred
65     Column  int    // Column (rune index) where the error
66     Err     error // The actual error
67 }
68
69 func (e *ParseError) Error() string {
70     return fmt.Sprintf("line %d, column %d: %s", e.Line,
71 }
72
73 // These are the errors that can be returned in ParseError.E
74 var (
75     ErrTrailingComma = errors.New("extra delimiter at en
76     ErrBareQuote    = errors.New("bare \" in non-quoted
77     ErrQuote        = errors.New("extraneous \" in fiel
78     ErrFieldCount   = errors.New("wrong number of field
79 )
80
81 // A Reader reads records from a CSV-encoded file.
82 //
83 // As returned by NewReader, a Reader expects input conformi
84 // The exported fields can be changed to customize the detai
85 // first call to Read or ReadAll.
86 //
87 // Comma is the field delimiter. It defaults to ','.
88 //
89 // Comment, if not 0, is the comment character. Lines beginn
90 // Comment character are ignored.
91 //

```

```

92 // If FieldsPerRecord is positive, Read requires each record
93 // have the given number of fields. If FieldsPerRecord is 0
94 // the number of fields in the first record, so that future
95 // have the same field count. If FieldsPerRecord is negativ
96 // made and records may have a variable number of fields.
97 //
98 // If LazyQuotes is true, a quote may appear in an unquoted
99 // non-doubled quote may appear in a quoted field.
100 //
101 // If TrailingComma is true, the last field may be an unquot
102 //
103 // If TrimLeadingSpace is true, leading white space in a fie
104 type Reader struct {
105     Comma      rune // Field delimiter (set to ',')
106     Comment    rune // Comment character for start
107     FieldsPerRecord int // Number of expected fields p
108     LazyQuotes bool // Allow lazy quotes
109     TrailingComma bool // Allow trailing comma
110     TrimLeadingSpace bool // Trim leading space
111     line       int
112     column     int
113     r          *bufio.Reader
114     field      bytes.Buffer
115 }
116
117 // NewReader returns a new Reader that reads from r.
118 func NewReader(r io.Reader) *Reader {
119     return &Reader{
120         Comma: ',',
121         r:     bufio.NewReader(r),
122     }
123 }
124
125 // error creates a new ParseError based on err.
126 func (r *Reader) error(err error) error {
127     return &ParseError{
128         Line:   r.line,
129         Column: r.column,
130         Err:    err,
131     }
132 }
133
134 // Read reads one record from r. The record is a slice of s
135 // string representing one field.
136 func (r *Reader) Read() (record []string, err error) {
137     for {
138         record, err = r.parseRecord()
139         if record != nil {
140             break

```

```

141         }
142         if err != nil {
143             return nil, err
144         }
145     }
146
147     if r.FieldsPerRecord > 0 {
148         if len(record) != r.FieldsPerRecord {
149             r.column = 0 // report at start of r
150             return record, r.error(ErrFieldCount
151         )
152     } else if r.FieldsPerRecord == 0 {
153         r.FieldsPerRecord = len(record)
154     }
155     return record, nil
156 }
157
158 // ReadAll reads all the remaining records from r.
159 // Each record is a slice of fields.
160 // A successful call returns err == nil, not err == EOF. Bec
161 // defined to read until EOF, it does not treat end of file
162 // reported.
163 func (r *Reader) ReadAll() (records [][]string, err error) {
164     for {
165         record, err := r.Read()
166         if err == io.EOF {
167             return records, nil
168         }
169         if err != nil {
170             return nil, err
171         }
172         records = append(records, record)
173     }
174     panic("unreachable")
175 }
176
177 // readRune reads one rune from r, folding \r\n to \n and ke
178 // of how far into the line we have read. r.column will poi
179 // of this rune, not the end of this rune.
180 func (r *Reader) readRune() (rune, error) {
181     r1, _, err := r.r.ReadRune()
182
183     // Handle \r\n here. We make the simplifying assump
184     // anytime \r is followed by \n that it can be folde
185     // We will not detect files which contain both \r\n
186     if r1 == '\r' {
187         r1, _, err = r.r.ReadRune()
188         if err == nil {
189             if r1 != '\n' {

```

```

190             r.r.UnreadRune()
191             r1 = '\r'
192         }
193     }
194 }
195     r.column++
196     return r1, err
197 }
198
199 // unreadRune puts the last rune read from r back.
200 func (r *Reader) unreadRune() {
201     r.r.UnreadRune()
202     r.column--
203 }
204
205 // skip reads runes up to and including the rune delim or un
206 func (r *Reader) skip(delim rune) error {
207     for {
208         r1, err := r.readRune()
209         if err != nil {
210             return err
211         }
212         if r1 == delim {
213             return nil
214         }
215     }
216     panic("unreachable")
217 }
218
219 // parseRecord reads and parses a single csv record from r.
220 func (r *Reader) parseRecord() (fields []string, err error) {
221     // Each record starts on a new line. We increment c
222     // number (lines start at 1, not 0) and set column t
223     // so as we increment in readRune it points to the c
224     r.line++
225     r.column = -1
226
227     // Peek at the first rune. If it is an error we are
228     // If we are support comments and it is the comment
229     // then skip to the end of line.
230
231     r1, _, err := r.r.ReadRune()
232     if err != nil {
233         return nil, err
234     }
235
236     if r.Comment != 0 && r1 == r.Comment {
237         return nil, r.skip('\n')
238     }
239     r.r.UnreadRune()

```

```

240
241 // At this point we have at least one field.
242 for {
243     haveField, delim, err := r.parseField()
244     if haveField {
245         fields = append(fields, r.field.Stri
246     }
247     if delim == '\n' || err == io.EOF {
248         return fields, err
249     } else if err != nil {
250         return nil, err
251     }
252 }
253 panic("unreachable")
254 }
255
256 // parseField parses the next field in the record. The read
257 // located in r.field. Delim is the first character not par
258 // (r.Comma or '\n').
259 func (r *Reader) parseField() (haveField bool, delim rune, e
260     r.field.Reset()
261
262     r1, err := r.readRune()
263     if err != nil {
264         // If we have EOF and are not at the start o
265         // then we return the empty field. We have
266         // checked for trailing commas if needed.
267         if err == io.EOF && r.column != 0 {
268             return true, 0, err
269         }
270         return false, 0, err
271     }
272
273     if r.TrimLeadingSpace {
274         for r1 != '\n' && unicode.IsSpace(r1) {
275             r1, err = r.readRune()
276             if err != nil {
277                 return false, 0, err
278             }
279         }
280     }
281
282     switch r1 {
283     case r.Comma:
284         // will check below
285
286     case '\n':
287         // We are a trailing empty field or a blank
288         if r.column == 0 {

```

```

289         return false, r1, nil
290     }
291     return true, r1, nil
292
293     case '':
294         // quoted field
295     Quoted:
296         for {
297             r1, err = r.readRune()
298             if err != nil {
299                 if err == io.EOF {
300                     if r.LazyQuotes {
301                         return true,
302                     }
303                     return false, 0, r.e
304                 }
305                 return false, 0, err
306             }
307             switch r1 {
308             case '':
309                 r1, err = r.readRune()
310                 if err != nil || r1 == r.Comma
311                     break Quoted
312             }
313             if r1 == '\n' {
314                 return true, r1, nil
315             }
316             if r1 != '' {
317                 if !r.LazyQuotes {
318                     r.column--
319                     return false
320                 }
321                 // accept the bare q
322                 r.field.WriteRune('
323             }
324             case '\n':
325                 r.line++
326                 r.column = -1
327             }
328             r.field.WriteRune(r1)
329         }
330
331     default:
332         // unquoted field
333         for {
334             r.field.WriteRune(r1)
335             r1, err = r.readRune()
336             if err != nil || r1 == r.Comma {
337                 break

```

```

338         }
339         if r1 == '\n' {
340             return true, r1, nil
341         }
342         if !r.LazyQuotes && r1 == '"' {
343             return false, 0, r.error(Err
344         }
345     }
346 }
347
348 if err != nil {
349     if err == io.EOF {
350         return true, 0, err
351     }
352     return false, 0, err
353 }
354
355 if !r.TrailingComma {
356     // We don't allow trailing commas. See if w
357     // are at the end of the line (being mindful
358     // of trimming spaces).
359     c := r.column
360     r1, err = r.readRune()
361     if r.TrimLeadingSpace {
362         for r1 != '\n' && unicode.IsSpace(r1
363             r1, err = r.readRune()
364             if err != nil {
365                 break
366             }
367         }
368     }
369     if err == io.EOF || r1 == '\n' {
370         r.column = c // report the comma
371         return false, 0, r.error(ErrTrailing
372     }
373     r.unreadRune()
374 }
375 return true, r1, nil
376 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/csv/writer.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package csv
6
7 import (
8     "bufio"
9     "io"
10    "strings"
11    "unicode"
12    "unicode/utf8"
13 )
14
15 // A Writer writes records to a CSV encoded file.
16 //
17 // As returned by NewWriter, a Writer writes records termina
18 // newline and uses ',' as the field delimiter. The exporte
19 // changed to customize the details before the first call to
20 //
21 // Comma is the field delimiter.
22 //
23 // If UseCRLF is true, the Writer ends each record with \r\n
24 type Writer struct {
25     Comma    rune // Field delimiter (set to to ',' by Ne
26     UseCRLF  bool // True to use \r\n as the line termina
27     w        *bufio.Writer
28 }
29
30 // NewWriter returns a new Writer that writes to w.
31 func NewWriter(w io.Writer) *Writer {
32     return &Writer{
33         Comma: ',',
34         w:     bufio.NewWriter(w),
35     }
36 }
37
38 // Writer writes a single CSV record to w along with any nec
39 // A record is a slice of strings with each string being one
40 func (w *Writer) Write(record []string) (err error) {
41     for n, field := range record {
```

```

42         if n > 0 {
43             if _, err = w.w.WriteRune(w.Comma);
44                 return
45             }
46         }
47
48         // If we don't have to have a quoted field t
49         // write out the field and continue to the n
50         if !w.fieldNeedsQuotes(field) {
51             if _, err = w.w.WriteString(field);
52                 return
53             }
54             continue
55         }
56         if err = w.w.WriteByte('"'); err != nil {
57             return
58         }
59
60         for _, r1 := range field {
61             switch r1 {
62                 case '"':
63                     _, err = w.w.WriteString(`"`)
64                 case '\r':
65                     if !w.UseCRLF {
66                         err = w.w.WriteByte(
67                             '\n')
68                     }
69                 case '\n':
70                     if w.UseCRLF {
71                         _, err = w.w.WriteSt
72                     } else {
73                         err = w.w.WriteByte(
74                             '\n')
75                     }
76                 default:
77                     _, err = w.w.WriteRune(r1)
78             }
79             if err != nil {
80                 return
81             }
82         }
83         if err = w.w.WriteByte('"'); err != nil {
84             return
85         }
86         if w.UseCRLF {
87             _, err = w.w.WriteString("\r\n")
88         } else {
89             err = w.w.WriteByte('\n')
90         }
91         return

```

```

92 }
93
94 // Flush writes any buffered data to the underlying io.Writer
95 func (w *Writer) Flush() {
96     w.w.Flush()
97 }
98
99 // WriteAll writes multiple CSV records to w using Write and
100 func (w *Writer) WriteAll(records [][]string) (err error) {
101     for _, record := range records {
102         err = w.Write(record)
103         if err != nil {
104             break
105         }
106     }
107     w.Flush()
108     return nil
109 }
110
111 // fieldNeedsQuotes returns true if our field must be enclosed
112 // Empty fields, fields with a Comma, fields with a quote or
113 // fields which start with a space must be enclosed in quote
114 func (w *Writer) fieldNeedsQuotes(field string) bool {
115     if len(field) == 0 || strings.IndexRune(field, w.Comma) >= 0 || strings.IndexRune(field, w.Quote) >= 0 || strings.IndexRune(field, ' ') >= 0 {
116         return true
117     }
118
119     r1, _ := utf8.DecodeRuneInString(field)
120     return unicode.IsSpace(r1)
121 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/gob/decode.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package gob
6
7 // TODO(rsc): When garbage collector changes, revisit
8 // the allocations in this file that use unsafe.Pointer.
9
10 import (
11     "bytes"
12     "errors"
13     "io"
14     "math"
15     "reflect"
16     "unsafe"
17 )
18
19 var (
20     errBadUint = errors.New("gob: encoded unsigned integ
21     errBadType = errors.New("gob: unknown type id or cor
22     errRange   = errors.New("gob: bad data: field number
23 )
24
25 // decoderState is the execution state of an instance of the
26 // is created for nested objects.
27 type decoderState struct {
28     dec *Decoder
29     // The buffer is stored with an extra indirection be
30     // if we load a type during decode (when reading an
31     b      *bytes.Buffer
32     fieldnum int // the last field number read.
33     buf     []byte
34     next   *decoderState // for free list
35 }
36
37 // We pass the bytes.Buffer separately for easier testing of
38 // without requiring a full Decoder.
39 func (dec *Decoder) newDecoderState(buf *bytes.Buffer) *deco
40     d := dec.freeList
41     if d == nil {
```

```

42             d = new(decoderState)
43             d.dec = dec
44             d.buf = make([]byte, uint64Size)
45         } else {
46             dec.freeList = d.next
47         }
48         d.b = buf
49         return d
50     }
51
52     func (dec *Decoder) freeDecoderState(d *decoderState) {
53         d.next = dec.freeList
54         dec.freeList = d
55     }
56
57     func overflow(name string) error {
58         return errors.New(`value for "` + name + `" out of r
59     }
60
61     // decodeUintReader reads an encoded unsigned integer from a
62     // Used only by the Decoder to read the message length.
63     func decodeUintReader(r io.Reader, buf []byte) (x uint64, wi
64         width = 1
65         _, err = r.Read(buf[0:width])
66         if err != nil {
67             return
68         }
69         b := buf[0]
70         if b <= 0x7f {
71             return uint64(b), width, nil
72         }
73         n := -int(int8(b))
74         if n > uint64Size {
75             err = errBadUint
76             return
77         }
78         width, err = io.ReadFull(r, buf[0:n])
79         if err != nil {
80             if err == io.EOF {
81                 err = io.ErrUnexpectedEOF
82             }
83             return
84         }
85         // Could check that the high byte is zero but it's n
86         for _, b := range buf[0:width] {
87             x = x<<8 | uint64(b)
88         }
89         width++ // +1 for length byte
90         return
91     }

```

```

92
93 // decodeUint reads an encoded unsigned integer from state.r
94 // Does not check for overflow.
95 func (state *decoderState) decodeUint() (x uint64) {
96     b, err := state.b.ReadByte()
97     if err != nil {
98         error_(err)
99     }
100    if b <= 0x7f {
101        return uint64(b)
102    }
103    n := -int(int8(b))
104    if n > uint64Size {
105        error_(errBadUint)
106    }
107    width, err := state.b.Read(state.buf[0:n])
108    if err != nil {
109        error_(err)
110    }
111    // Don't need to check error; it's safe to loop rega
112    // Could check that the high byte is zero but it's n
113    for _, b := range state.buf[0:width] {
114        x = x<<8 | uint64(b)
115    }
116    return x
117 }
118
119 // decodeInt reads an encoded signed integer from state.r.
120 // Does not check for overflow.
121 func (state *decoderState) decodeInt() int64 {
122     x := state.decodeUint()
123     if x&1 != 0 {
124         return ^int64(x >> 1)
125     }
126     return int64(x >> 1)
127 }
128
129 // decOp is the signature of a decoding operator for a given
130 type decOp func(i *decInstr, state *decoderState, p unsafe.P
131
132 // The 'instructions' of the decoding machine
133 type decInstr struct {
134     op      decOp
135     field  int    // field number of the wire type
136     indir  int    // how many pointer indirections to r
137     offset uintptr // offset in the structure of the fie
138     ovfl   error   // error message for overflow/underfl
139 }
140

```

```

141 // Since the encoder writes no zeros, if we arrive at a deco
142 // a value to extract and store. The field number has already
143 // (it's how we knew to call this decoder).
144 // Each decoder is responsible for handling any indirections
145 // with the data structure. If any pointer so reached is not
146 // be done.
147
148 // Walk the pointer hierarchy, allocating if we find a nil.
149 func decIndirect(p unsafe.Pointer, indir int) unsafe.Pointer
150     for ; indir > 1; indir-- {
151         if *(*unsafe.Pointer)(p) == nil {
152             // Allocation required
153             *(*unsafe.Pointer)(p) = unsafe.Pointer{0}
154         }
155         p = *(*unsafe.Pointer)(p)
156     }
157     return p
158 }
159
160 // ignoreUint discards a uint value with no destination.
161 func ignoreUint(i *decInstr, state *decoderState, p unsafe.Pointer)
162     state.decodeUint()
163 }
164
165 // ignoreTwoUints discards a uint value with no destination.
166 // complex values.
167 func ignoreTwoUints(i *decInstr, state *decoderState, p unsafe.Pointer)
168     state.decodeUint()
169     state.decodeUint()
170 }
171
172 // decBool decodes a uint and stores it as a boolean through
173 func decBool(i *decInstr, state *decoderState, p unsafe.Pointer)
174     if i.indir > 0 {
175         if *(*unsafe.Pointer)(p) == nil {
176             *(*unsafe.Pointer)(p) = unsafe.Pointer{0}
177         }
178         p = *(*unsafe.Pointer)(p)
179     }
180     *(*bool)(p) = state.decodeUint() != 0
181 }
182
183 // decInt8 decodes an integer and stores it as an int8 through
184 func decInt8(i *decInstr, state *decoderState, p unsafe.Pointer)
185     if i.indir > 0 {
186         if *(*unsafe.Pointer)(p) == nil {
187             *(*unsafe.Pointer)(p) = unsafe.Pointer{0}
188         }
189         p = *(*unsafe.Pointer)(p)

```

```

190     }
191     v := state.decodeInt()
192     if v < math.MinInt8 || math.MaxInt8 < v {
193         error_(i.ovfl)
194     } else {
195         *(*int8)(p) = int8(v)
196     }
197 }
198
199 // decUint8 decodes an unsigned integer and stores it as a u
200 func decUint8(i *decInstr, state *decoderState, p unsafe.Poi
201     if i.indir > 0 {
202         if *(*unsafe.Pointer)(p) == nil {
203             *(*unsafe.Pointer)(p) = unsafe.Point
204         }
205         p = *(*unsafe.Pointer)(p)
206     }
207     v := state.decodeUint()
208     if math.MaxUint8 < v {
209         error_(i.ovfl)
210     } else {
211         *(*uint8)(p) = uint8(v)
212     }
213 }
214
215 // decInt16 decodes an integer and stores it as an int16 thr
216 func decInt16(i *decInstr, state *decoderState, p unsafe.Poi
217     if i.indir > 0 {
218         if *(*unsafe.Pointer)(p) == nil {
219             *(*unsafe.Pointer)(p) = unsafe.Point
220         }
221         p = *(*unsafe.Pointer)(p)
222     }
223     v := state.decodeInt()
224     if v < math.MinInt16 || math.MaxInt16 < v {
225         error_(i.ovfl)
226     } else {
227         *(*int16)(p) = int16(v)
228     }
229 }
230
231 // decUint16 decodes an unsigned integer and stores it as a
232 func decUint16(i *decInstr, state *decoderState, p unsafe.Po
233     if i.indir > 0 {
234         if *(*unsafe.Pointer)(p) == nil {
235             *(*unsafe.Pointer)(p) = unsafe.Point
236         }
237         p = *(*unsafe.Pointer)(p)
238     }
239     v := state.decodeUint()

```

```

240         if math.MaxUint16 < v {
241             error_(i.ovfl)
242         } else {
243             *(*uint16)(p) = uint16(v)
244         }
245     }
246
247 // decInt32 decodes an integer and stores it as an int32 thr
248 func decInt32(i *decInstr, state *decoderState, p unsafe.Poi
249     if i.indir > 0 {
250         if *(*unsafe.Pointer)(p) == nil {
251             *(*unsafe.Pointer)(p) = unsafe.Point
252         }
253         p = *(*unsafe.Pointer)(p)
254     }
255     v := state.decodeInt()
256     if v < math.MinInt32 || math.MaxInt32 < v {
257         error_(i.ovfl)
258     } else {
259         *(*int32)(p) = int32(v)
260     }
261 }
262
263 // decUint32 decodes an unsigned integer and stores it as a
264 func decUint32(i *decInstr, state *decoderState, p unsafe.Po
265     if i.indir > 0 {
266         if *(*unsafe.Pointer)(p) == nil {
267             *(*unsafe.Pointer)(p) = unsafe.Point
268         }
269         p = *(*unsafe.Pointer)(p)
270     }
271     v := state.decodeUint()
272     if math.MaxUint32 < v {
273         error_(i.ovfl)
274     } else {
275         *(*uint32)(p) = uint32(v)
276     }
277 }
278
279 // decInt64 decodes an integer and stores it as an int64 thr
280 func decInt64(i *decInstr, state *decoderState, p unsafe.Poi
281     if i.indir > 0 {
282         if *(*unsafe.Pointer)(p) == nil {
283             *(*unsafe.Pointer)(p) = unsafe.Point
284         }
285         p = *(*unsafe.Pointer)(p)
286     }
287     *(*int64)(p) = int64(state.decodeInt())
288 }

```

```

289
290 // decUint64 decodes an unsigned integer and stores it as a
291 func decUint64(i *decInstr, state *decoderState, p unsafe.Pointer) {
292     if i.indir > 0 {
293         if *(*unsafe.Pointer)(p) == nil {
294             *(*unsafe.Pointer)(p) = unsafe.Pointer{}
295         }
296         p = *(*unsafe.Pointer)(p)
297     }
298     *(*uint64)(p) = uint64(state.decodeUint())
299 }
300
301 // Floating-point numbers are transmitted as uint64s holding
302 // of the underlying representation. They are sent byte-rev
303 // the exponent end coming out first, so integer floating point
304 // (for example) transmit more compactly. This routine does
305 // unswizzling.
306 func floatFromBits(u uint64) float64 {
307     var v uint64
308     for i := 0; i < 8; i++ {
309         v <<= 8
310         v |= u & 0xFF
311         u >>= 8
312     }
313     return math.Float64frombits(v)
314 }
315
316 // storeFloat32 decodes an unsigned integer, treats it as a
317 // number, and stores it through p. It's a helper function for
318 func storeFloat32(i *decInstr, state *decoderState, p unsafe.Pointer) {
319     v := floatFromBits(state.decodeUint())
320     av := v
321     if av < 0 {
322         av = -av
323     }
324     // +Inf is OK in both 32- and 64-bit floats. Underflow
325     if math.MaxFloat32 < av && av <= math.MaxFloat64 {
326         error_(i.ovfl)
327     } else {
328         *(*float32)(p) = float32(v)
329     }
330 }
331
332 // decFloat32 decodes an unsigned integer, treats it as a 32
333 // number, and stores it through p.
334 func decFloat32(i *decInstr, state *decoderState, p unsafe.Pointer) {
335     if i.indir > 0 {
336         if *(*unsafe.Pointer)(p) == nil {
337             *(*unsafe.Pointer)(p) = unsafe.Pointer{}

```

```

338         }
339         p = *(*unsafe.Pointer)(p)
340     }
341     storeFloat32(i, state, p)
342 }
343
344 // decFloat64 decodes an unsigned integer, treats it as a 64
345 // number, and stores it through p.
346 func decFloat64(i *decInstr, state *decoderState, p unsafe.P
347     if i.indir > 0 {
348         if *(*unsafe.Pointer)(p) == nil {
349             *(*unsafe.Pointer)(p) = unsafe.Point
350         }
351         p = *(*unsafe.Pointer)(p)
352     }
353     *(*float64)(p) = floatFromBits(uint64(state.decodeUi
354 }
355
356 // decComplex64 decodes a pair of unsigned integers, treats
357 // pair of floating point numbers, and stores them as a comp
358 // The real part comes first.
359 func decComplex64(i *decInstr, state *decoderState, p unsafe
360     if i.indir > 0 {
361         if *(*unsafe.Pointer)(p) == nil {
362             *(*unsafe.Pointer)(p) = unsafe.Point
363         }
364         p = *(*unsafe.Pointer)(p)
365     }
366     storeFloat32(i, state, p)
367     storeFloat32(i, state, unsafe.Pointer(uintptr(p)+uns
368 }
369
370 // decComplex128 decodes a pair of unsigned integers, treats
371 // pair of floating point numbers, and stores them as a comp
372 // The real part comes first.
373 func decComplex128(i *decInstr, state *decoderState, p unsaf
374     if i.indir > 0 {
375         if *(*unsafe.Pointer)(p) == nil {
376             *(*unsafe.Pointer)(p) = unsafe.Point
377         }
378         p = *(*unsafe.Pointer)(p)
379     }
380     real := floatFromBits(uint64(state.decodeUint()))
381     imag := floatFromBits(uint64(state.decodeUint()))
382     *(*complex128)(p) = complex(real, imag)
383 }
384
385 // decUint8Slice decodes a byte slice and stores through p a
386 // describing the data.
387 // uint8 slices are encoded as an unsigned count followed by

```

```

388 func decUint8Slice(i *decInstr, state *decoderState, p unsafe
389     if i.indir > 0 {
390         if *(*unsafe.Pointer)(p) == nil {
391             *(*unsafe.Pointer)(p) = unsafe.Pointer
392         }
393         p = *(*unsafe.Pointer)(p)
394     }
395     n := state.decodeUint()
396     if n > uint64(state.b.Len()) {
397         errorf("length of []byte exceeds input size
398     }
399     slice := ([]uint8)(p)
400     if uint64(cap(*slice)) < n {
401         *slice = make([]uint8, n)
402     } else {
403         *slice = (*slice)[0:n]
404     }
405     if _, err := state.b.Read(*slice); err != nil {
406         errorf("error decoding []byte: %s", err)
407     }
408 }
409
410 // decString decodes byte array and stores through p a string
411 // describing the data.
412 // Strings are encoded as an unsigned count followed by the
413 func decString(i *decInstr, state *decoderState, p unsafe.Pointer)
414     if i.indir > 0 {
415         if *(*unsafe.Pointer)(p) == nil {
416             *(*unsafe.Pointer)(p) = unsafe.Pointer
417         }
418         p = *(*unsafe.Pointer)(p)
419     }
420     n := state.decodeUint()
421     if n > uint64(state.b.Len()) {
422         errorf("string length exceeds input size (%d
423     }
424     b := make([]byte, n)
425     state.b.Read(b)
426     // It would be a shame to do the obvious thing here,
427     //     *(*string)(p) = string(b)
428     // because we've already allocated the storage and then
429     // allocate again and copy. So we do this ugly hack
430     // even more unsafe than it looks as it depends on the
431     // representation of a string matching the beginning
432     // representation of a byte slice (a byte slice is 1
433     *(*string)(p) = *(*string)(unsafe.Pointer(&b))
434 }
435
436 // ignoreUint8Array skips over the data for a byte slice val

```

```

437 func ignoreUint8Array(i *decInstr, state *decoderState, p un
438     b := make([]byte, state.decodeUint())
439     state.b.Read(b)
440 }
441
442 // Execution engine
443
444 // The encoder engine is an array of instructions indexed by
445 // decoder. It is executed with random access according to
446 type decEngine struct {
447     instr []decInstr
448     numInstr int // the number of active instructions
449 }
450
451 // allocate makes sure storage is available for an object of
452 // that is indir levels of indirection through p.
453 func allocate(rtyp reflect.Type, p uintptr, indir int) uintptr
454     if indir == 0 {
455         return p
456     }
457     up := unsafe.Pointer(p)
458     if indir > 1 {
459         up = decIndirect(up, indir)
460     }
461     if>(*unsafe.Pointer)(up) == nil {
462         // Allocate object.
463        >(*unsafe.Pointer)(up) = unsafe.Pointer(refl
464     }
465     return>(*uintptr)(up)
466 }
467
468 // decodeSingle decodes a top-level value that is not a stru
469 // Such values are preceded by a zero, making them have the
470 // struct field (although with an illegal field number).
471 func (dec *Decoder) decodeSingle(engine *decEngine, ut *user
472     state := dec.newDecoderState(&dec.buf)
473     state.fieldnum = singletonField
474     delta := int(state.decodeUint())
475     if delta != 0 {
476         errorf("decode: corrupted data: non-zero del
477     }
478     instr := &engine.instr[singletonField]
479     if instr.indir != ut.indir {
480         errorf("internal error: inconsistent indirec
481     }
482     ptr := unsafe.Pointer(basep) // offset will be zero
483     if instr.indir > 1 {
484         ptr = decIndirect(ptr, instr.indir)
485     }

```

```

486         instr.op(instr, state, ptr)
487         dec.freeDecoderState(state)
488     }
489
490     // decodeStruct decodes a top-level struct and stores it thr
491     // Indir is for the value, not the type. At the time of the
492     // differ from ut.indir, which was computed when the engine
493     // This state cannot arise for decodeSingle, which is called
494     // from the user's value, not from the innards of an engine.
495     func (dec *Decoder) decodeStruct(engine *decEngine, ut *user
496         p = allocate(ut.base, p, indir)
497         state := dec.newDecoderState(&dec.buf)
498         state.fieldnum = -1
499         basep := p
500         for state.b.Len() > 0 {
501             delta := int(state.decodeUint())
502             if delta < 0 {
503                 errorf("decode: corrupted data: nega
504             }
505             if delta == 0 { // struct terminator is zero
506                 break
507             }
508             fieldnum := state.fieldnum + delta
509             if fieldnum >= len(engine.instr) {
510                 error_(errRange)
511                 break
512             }
513             instr := &engine.instr[fieldnum]
514             p := unsafe.Pointer(basep + instr.offset)
515             if instr.indir > 1 {
516                 p = decIndirect(p, instr.indir)
517             }
518             instr.op(instr, state, p)
519             state.fieldnum = fieldnum
520         }
521         dec.freeDecoderState(state)
522     }
523
524     // ignoreStruct discards the data for a struct with no desti
525     func (dec *Decoder) ignoreStruct(engine *decEngine) {
526         state := dec.newDecoderState(&dec.buf)
527         state.fieldnum = -1
528         for state.b.Len() > 0 {
529             delta := int(state.decodeUint())
530             if delta < 0 {
531                 errorf("ignore decode: corrupted dat
532             }
533             if delta == 0 { // struct terminator is zero
534                 break
535             }

```

```

536         fieldnum := state.fieldnum + delta
537         if fieldnum >= len(engine.instr) {
538             error_(errRange)
539         }
540         instr := &engine.instr[fieldnum]
541         instr.op(instr, state, unsafe.Pointer(nil))
542         state.fieldnum = fieldnum
543     }
544     dec.freeDecoderState(state)
545 }
546
547 // ignoreSingle discards the data for a top-level non-struct
548 // destination. It's used when calling Decode with a nil val
549 func (dec *Decoder) ignoreSingle(engine *decEngine) {
550     state := dec.newDecoderState(&dec.buf)
551     state.fieldnum = singletonField
552     delta := int(state.decodeUint())
553     if delta != 0 {
554         errorf("decode: corrupted data: non-zero del
555     }
556     instr := &engine.instr[singletonField]
557     instr.op(instr, state, unsafe.Pointer(nil))
558     dec.freeDecoderState(state)
559 }
560
561 // decodeArrayHelper does the work for decoding arrays and s
562 func (dec *Decoder) decodeArrayHelper(state *decoderState, p
563     instr := &decInstr{elemOp, 0, elemIndir, 0, ovfl}
564     for i := 0; i < length; i++ {
565         up := unsafe.Pointer(p)
566         if elemIndir > 1 {
567             up = decIndirect(up, elemIndir)
568         }
569         elemOp(instr, state, up)
570         p += uintptr(elemWid)
571     }
572 }
573
574 // decodeArray decodes an array and stores it through p, tha
575 // The length is an unsigned integer preceding the elements.
576 // (it's part of the type), it's a useful check and is inclu
577 func (dec *Decoder) decodeArray(atyp reflect.Type, state *de
578     if indir > 0 {
579         p = allocate(atyp, p, 1) // All but the last
580     }
581     if n := state.decodeUint(); n != uint64(length) {
582         errorf("length mismatch in decodeArray")
583     }
584     dec.decodeArrayHelper(state, p, elemOp, elemWid, len

```

```

585 }
586
587 // decodeIntoValue is a helper for map decoding. Since maps
588 // unlike the other items we can't use a pointer directly.
589 func decodeIntoValue(state *decoderState, op decOp, indir in
590     instr := &decInstr{op, 0, indir, 0, ovfl}
591     up := unsafe.Pointer(unsafeAddr(v))
592     if indir > 1 {
593         up = decIndirect(up, indir)
594     }
595     op(instr, state, up)
596     return v
597 }
598
599 // decodeMap decodes a map and stores its header through p.
600 // Maps are encoded as a length followed by key:value pairs.
601 // Because the internals of maps are not visible to us, we m
602 // use reflection rather than pointer magic.
603 func (dec *Decoder) decodeMap(mtyp reflect.Type, state *deco
604     if indir > 0 {
605         p = allocate(mtyp, p, 1) // All but the last
606     }
607     up := unsafe.Pointer(p)
608     if *(*unsafe.Pointer)(up) == nil { // maps are repre
609         // Allocate map.
610         *(*unsafe.Pointer)(up) = unsafe.Pointer(refl
611     }
612     // Maps cannot be accessed by moving addresses aroun
613     // that slices etc. can. We must recover a full ref
614     // the iteration.
615     v := reflect.NewAt(mtyp, unsafe.Pointer(p)).Elem()
616     n := int(state.decodeUint())
617     for i := 0; i < n; i++ {
618         key := decodeIntoValue(state, keyOp, keyIndi
619         elem := decodeIntoValue(state, elemOp, elemI
620         v.SetMapIndex(key, elem)
621     }
622 }
623
624 // ignoreArrayHelper does the work for discarding arrays and
625 func (dec *Decoder) ignoreArrayHelper(state *decoderState, e
626     instr := &decInstr{elemOp, 0, 0, 0, errors.New("no e
627     for i := 0; i < length; i++ {
628         elemOp(instr, state, nil)
629     }
630 }
631
632 // ignoreArray discards the data for an array value with no
633 func (dec *Decoder) ignoreArray(state *decoderState, elemOp

```

```

634         if n := state.decodeUint(); n != uint64(length) {
635             errorf("length mismatch in ignoreArray")
636         }
637         dec.ignoreArrayHelper(state, elemOp, length)
638     }
639
640 // ignoreMap discards the data for a map value with no desti
641 func (dec *Decoder) ignoreMap(state *decoderState, keyOp, el
642     n := int(state.decodeUint())
643     keyInstr := &decInstr{keyOp, 0, 0, 0, errors.New("no
644     elemInstr := &decInstr{elemOp, 0, 0, 0, errors.New("
645     for i := 0; i < n; i++ {
646         keyOp(keyInstr, state, nil)
647         elemOp(elemInstr, state, nil)
648     }
649 }
650
651 // decodeSlice decodes a slice and stores the slice header t
652 // Slices are encoded as an unsigned length followed by the
653 func (dec *Decoder) decodeSlice(atyp reflect.Type, state *de
654     nr := state.decodeUint()
655     if nr > uint64(state.b.Len()) {
656         errorf("length of slice exceeds input size (
657     }
658     n := int(nr)
659     if indir > 0 {
660         up := unsafe.Pointer(p)
661         if *(*unsafe.Pointer)(up) == nil {
662             // Allocate the slice header.
663             *(*unsafe.Pointer)(up) = unsafe.Poin
664         }
665         p = *(*uintptr)(up)
666     }
667     // Allocate storage for the slice elements, that is,
668     // if the existing slice does not have the capacity.
669     // Always write a header at p.
670     hdrp := (*reflect.SliceHeader)(unsafe.Pointer(p))
671     if hdrp.Cap < n {
672         hdrp.Data = reflect.MakeSlice(atyp, n, n).Po
673         hdrp.Cap = n
674     }
675     hdrp.Len = n
676     dec.decodeArrayHelper(state, hdrp.Data, elemOp, elem
677 }
678
679 // ignoreSlice skips over the data for a slice value with no
680 func (dec *Decoder) ignoreSlice(state *decoderState, elemOp
681     dec.ignoreArrayHelper(state, elemOp, int(state.decod
682 }
683

```

```

684 // setInterfaceValue sets an interface value to a concrete v
685 // but first it checks that the assignment will succeed.
686 func setInterfaceValue(ivalue reflect.Value, value reflect.V
687     if !value.Type().AssignableTo(ivalue.Type()) {
688         errorf("cannot assign value of type %s to %s
689     }
690     ivalue.Set(value)
691 }
692
693 // decodeInterface decodes an interface value and stores it
694 // Interfaces are encoded as the name of a concrete type fol
695 // If the name is empty, the value is nil and no value is se
696 func (dec *Decoder) decodeInterface(ityp reflect.Type, state
697     // Create a writable interface reflect.Value. We ne
698     ivalue := allocValue(ityp)
699     // Read the name of the concrete type.
700     nr := state.decodeUint()
701     if nr < 0 || nr > 1<<31 { // zero is permissible for
702         errorf("invalid type name length %d", nr)
703     }
704     b := make([]byte, nr)
705     state.b.Read(b)
706     name := string(b)
707     if name == "" {
708         // Copy the representation of the nil interf
709         // This is horribly unsafe and special.
710         if indir > 0 {
711             p = allocate(ityp, p, 1) // All but
712         }
713         *(*[2]uintptr)(unsafe.Pointer(p)) = ivalue.I
714         return
715     }
716     if len(name) > 1024 {
717         errorf("name too long (%d bytes): %.20q...",
718     }
719     // The concrete type must be registered.
720     typ, ok := nameToConcreteType[name]
721     if !ok {
722         errorf("name not registered for interface: %
723     }
724     // Read the type id of the concrete value.
725     concreteId := dec.decodeTypeSequence(true)
726     if concreteId < 0 {
727         error_(dec.err)
728     }
729     // Byte count of value is next; we don't care what i
730     // in case we want to ignore the value by skipping i
731     state.decodeUint()
732     // Read the concrete value.

```

```

733     value := allocValue(typ)
734     dec.decodeValue(concreteId, value)
735     if dec.err != nil {
736         error_(dec.err)
737     }
738     // Allocate the destination interface value.
739     if indir > 0 {
740         p = allocate(ityp, p, 1) // All but the last
741     }
742     // Assign the concrete value to the interface.
743     // Tread carefully; it might not satisfy the interfa
744     setInterfaceValue(ivalue, value)
745     // Copy the representation of the interface value to
746     // This is horribly unsafe and special.
747     *(*[2]uintptr)(unsafe.Pointer(p)) = ivalue.Interface
748 }
749
750 // ignoreInterface discards the data for an interface value
751 func (dec *Decoder) ignoreInterface(state *decoderState) {
752     // Read the name of the concrete type.
753     b := make([]byte, state.decodeUint())
754     _, err := state.b.Read(b)
755     if err != nil {
756         error_(err)
757     }
758     id := dec.decodeTypeSequence(true)
759     if id < 0 {
760         error_(dec.err)
761     }
762     // At this point, the decoder buffer contains a deli
763     state.b.Next(int(state.decodeUint()))
764 }
765
766 // decodeGobDecoder decodes something implementing the GobDe
767 // The data is encoded as a byte slice.
768 func (dec *Decoder) decodeGobDecoder(state *decoderState, v
769     // Read the bytes for the value.
770     b := make([]byte, state.decodeUint())
771     _, err := state.b.Read(b)
772     if err != nil {
773         error_(err)
774     }
775     // We know it's a GobDecoder, so just call the metho
776     err = v.Interface().(GobDecoder).GobDecode(b)
777     if err != nil {
778         error_(err)
779     }
780 }
781

```

```

782 // ignoreGobDecoder discards the data for a GobDecoder value
783 func (dec *Decoder) ignoreGobDecoder(state *decoderState) {
784     // Read the bytes for the value.
785     b := make([]byte, state.decodeUint())
786     _, err := state.b.Read(b)
787     if err != nil {
788         error_(err)
789     }
790 }
791
792 // Index by Go types.
793 var decOpTable = [...]decOp{
794     reflect.Bool:      decBool,
795     reflect.Int8:      decInt8,
796     reflect.Int16:     decInt16,
797     reflect.Int32:     decInt32,
798     reflect.Int64:     decInt64,
799     reflect.Uint8:     decUint8,
800     reflect.Uint16:    decUint16,
801     reflect.Uint32:    decUint32,
802     reflect.Uint64:    decUint64,
803     reflect.Float32:   decFloat32,
804     reflect.Float64:   decFloat64,
805     reflect.Complex64: decComplex64,
806     reflect.Complex128: decComplex128,
807     reflect.String:    decString,
808 }
809
810 // Indexed by gob types. tComplex will be added during type
811 var decIgnoreOpMap = map[typeId]decOp{
812     tBool:    ignoreUint,
813     tInt:     ignoreUint,
814     tUint:    ignoreUint,
815     tFloat:   ignoreUint,
816     tBytes:   ignoreUint8Array,
817     tString:  ignoreUint8Array,
818     tComplex: ignoreTwoUints,
819 }
820
821 // decOpFor returns the decoding op for the base type under
822 // the indirection count to reach it.
823 func (dec *Decoder) decOpFor(wireId typeId, rt reflect.Type,
824     ut := userType(rt)
825     // If the type implements GobEncoder, we handle it w
826     if ut.isGobDecoder {
827         return dec.gobDecodeOpFor(ut)
828     }
829     // If this type is already in progress, it's a recur
830     // Return the pointer to the op we're already buildi
831     if opPtr := inProgress[rt]; opPtr != nil {

```

```

832         return opPtr, ut.indir
833     }
834     typ := ut.base
835     indir := ut.indir
836     var op decOp
837     k := typ.Kind()
838     if int(k) < len(decOpTable) {
839         op = decOpTable[k]
840     }
841     if op == nil {
842         inProgress[rt] = &op
843         // Special cases
844         switch t := typ; t.Kind() {
845             case reflect.Array:
846                 name = "element of " + name
847                 elemId := dec.wireType[wireId].Array
848                 elemOp, elemIndir := dec.decOpFor(el
849                 ovfl := overflow(name)
850                 op = func(i *decInstr, state *decode
851                     state.dec.decodeArray(t, sta
852                 }
853
854             case reflect.Map:
855                 keyId := dec.wireType[wireId].MapT.K
856                 elemId := dec.wireType[wireId].MapT.
857                 keyOp, keyIndir := dec.decOpFor(keyI
858                 elemOp, elemIndir := dec.decOpFor(el
859                 ovfl := overflow(name)
860                 op = func(i *decInstr, state *decode
861                     up := unsafe.Pointer(p)
862                     state.dec.decodeMap(t, state
863                 }
864
865             case reflect.Slice:
866                 name = "element of " + name
867                 if t.Elem().Kind() == reflect.Uint8
868                     op = decUint8Slice
869                     break
870                 }
871                 var elemId typeId
872                 if tt, ok := builtinIdToType[wireId]
873                     elemId = tt.(*sliceType).Ele
874                 } else {
875                     elemId = dec.wireType[wireId
876                 }
877                 elemOp, elemIndir := dec.decOpFor(el
878                 ovfl := overflow(name)
879                 op = func(i *decInstr, state *decode
880                     state.dec.decodeSlice(t, sta

```



```

930         keyId := dec.wireType[wireId].MapT.K
931         elemId := dec.wireType[wireId].MapT.
932         keyOp := dec.decIgnoreOpFor(keyId)
933         elemOp := dec.decIgnoreOpFor(elemId)
934         op = func(i *decInstr, state *decode
935                 state.dec.ignoreMap(state, k
936                 }
937
938     case wire.SliceT != nil:
939         elemId := wire.SliceT.Elem
940         elemOp := dec.decIgnoreOpFor(elemId)
941         op = func(i *decInstr, state *decode
942                 state.dec.ignoreSlice(state,
943                 }
944
945     case wire.StructT != nil:
946         // Generate a closure that calls out
947         enginePtr, err := dec.getIgnoreEngin
948         if err != nil {
949             error_(err)
950         }
951         op = func(i *decInstr, state *decode
952                 // indirect through enginePt
953                 state.dec.ignoreStruct(*engi
954                 }
955
956     case wire.GobEncoderT != nil:
957         op = func(i *decInstr, state *decode
958                 state.dec.ignoreGobDecoder(s
959                 }
960     }
961 }
962 if op == nil {
963     errorf("bad data: ignore can't handle type %
964 }
965 return op
966 }
967
968 // gobDecodeOpFor returns the op for a type that is known to
969 // GobDecoder.
970 func (dec *Decoder) gobDecodeOpFor(ut *userTypeInfo) (*decOp
971     rcvrType := ut.user
972     if ut.decIndir == -1 {
973         rcvrType = reflect.PtrTo(rcvrType)
974     } else if ut.decIndir > 0 {
975         for i := int8(0); i < ut.decIndir; i++ {
976             rcvrType = rcvrType.Elem()
977         }
978     }
979     var op decOp

```

```

980     op = func(i *decInstr, state *decoderState, p unsafe
981         // Caller has gotten us to within one indire
982         if i.indir > 0 {
983             if *(*unsafe.Pointer)(p) == nil {
984                 *(*unsafe.Pointer)(p) = unsa
985             }
986         }
987         // Now p is a pointer to the base type. Do
988         // get to the receiver type?
989         var v reflect.Value
990         if ut.decIndir == -1 {
991             v = reflect.NewAt(rcvrType, unsafe.P
992         } else {
993             v = reflect.NewAt(rcvrType, p).Elem(
994         }
995         state.dec.decodeGobDecoder(state, v)
996     }
997     return &op, int(ut.indir)
998 }
999 }
1000
1001 // compatibleType asks: Are these two gob Types compatible?
1002 // Answers the question for basic types, arrays, maps and sl
1003 // GobEncoder/Decoder pairs.
1004 // Structs are considered ok; fields will be checked later.
1005 func (dec *Decoder) compatibleType(fr reflect.Type, fw typeI
1006     if rhs, ok := inProgress[fr]; ok {
1007         return rhs == fw
1008     }
1009     inProgress[fr] = fw
1010     ut := userType(fr)
1011     wire, ok := dec.wireType[fw]
1012     // If fr is a GobDecoder, the wire type must be GobE
1013     // And if fr is not a GobDecoder, the wire type must
1014     if ut.isGobDecoder != (ok && wire.GobEncoderT != nil
1015         return false
1016     }
1017     if ut.isGobDecoder { // This test trumps all others.
1018         return true
1019     }
1020     switch t := ut.base; t.Kind() {
1021     default:
1022         // chan, etc: cannot handle.
1023         return false
1024     case reflect.Bool:
1025         return fw == tBool
1026     case reflect.Int, reflect.Int8, reflect.Int16, refle
1027         return fw == tInt
1028     case reflect.Uint, reflect.Uint8, reflect.Uint16, re

```

```

1029         return fw == tUint
1030     case reflect.Float32, reflect.Float64:
1031         return fw == tFloat
1032     case reflect.Complex64, reflect.Complex128:
1033         return fw == tComplex
1034     case reflect.String:
1035         return fw == tString
1036     case reflect.Interface:
1037         return fw == tInterface
1038     case reflect.Array:
1039         if !ok || wire.ArrayT == nil {
1040             return false
1041         }
1042         array := wire.ArrayT
1043         return t.Len() == array.Len && dec.compatible
1044     case reflect.Map:
1045         if !ok || wire.MapT == nil {
1046             return false
1047         }
1048         MapType := wire.MapT
1049         return dec.compatibleType(t.Key(), MapType.K)
1050     case reflect.Slice:
1051         // Is it an array of bytes?
1052         if t.Elem().Kind() == reflect.Uint8 {
1053             return fw == tBytes
1054         }
1055         // Extract and compare element types.
1056         var sw *sliceType
1057         if tt, ok := builtinIdToType[fw]; ok {
1058             sw, _ = tt.(*sliceType)
1059         } else if wire != nil {
1060             sw = wire.SliceT
1061         }
1062         elem := userType(t.Elem()).base
1063         return sw != nil && dec.compatibleType(elem,
1064     case reflect.Struct:
1065         return true
1066     }
1067     return true
1068 }
1069
1070 // typeString returns a human-readable description of the ty
1071 func (dec *Decoder) typeString(remoteId typeId) string {
1072     if t := idToType[remoteId]; t != nil {
1073         // globally known type.
1074         return t.string()
1075     }
1076     return dec.wireType[remoteId].string()
1077 }

```

```

1078
1079 // compileSingle compiles the decoder engine for a non-struct
1080 // GobDecoders.
1081 func (dec *Decoder) compileSingle(remoteId typeId, ut *userType
1082     rt := ut.user
1083     engine = new(decEngine)
1084     engine.instr = make([]decInstr, 1) // one item
1085     name := rt.String() // best we can do
1086     if !dec.compatibleType(rt, remoteId, make(map[reflect.Type]decInstr{
1087         remoteType := dec.typeString(remoteId)
1088         // Common confusing case: local interface type
1089         if ut.base.Kind() == reflect.Interface && remoteType == rt.Type() {
1090             return nil, errors.New("gob: local interface type not supported")
1091         }
1092     }) {
1093         return nil, errors.New("gob: decoding into local interface type not supported")
1094     }
1095     op, indir := dec.decOpFor(remoteId, rt, name, make(map[reflect.Type]decInstr{
1096         remoteType := dec.typeString(remoteId)
1097         // Common confusing case: local interface type
1098         if ut.base.Kind() == reflect.Interface && remoteType == rt.Type() {
1099             return nil, errors.New("gob: local interface type not supported")
1100         }
1101     }) {
1102         return nil, errors.New("gob: decoding into local interface type not supported")
1103     }
1104     ovfl := errors.New(`value for "` + name + `" out of range`)
1105     engine.instr[singletonField] = decInstr{*op, singletonField, 0, 0, ovfl}
1106     engine.numInstr = 1
1107     return
1108 }
1109
1110 // compileIgnoreSingle compiles the decoder engine for a non-struct
1111 func (dec *Decoder) compileIgnoreSingle(remoteId typeId) (engine *decEngine) {
1112     engine = new(decEngine)
1113     engine.instr = make([]decInstr, 1) // one item
1114     op := dec.decIgnoreOpFor(remoteId)
1115     ovfl := overflow(dec.typeString(remoteId))
1116     engine.instr[0] = decInstr{op, 0, 0, 0, ovfl}
1117     engine.numInstr = 1
1118     return
1119 }
1120
1121 // compileDec compiles the decoder engine for a value. If it
1122 // it calls out to compileSingle.
1123 func (dec *Decoder) compileDec(remoteId typeId, ut *userType) {
1124     rt := ut.base
1125     srt := rt
1126     if srt.Kind() != reflect.Struct || !ut.isGobDecoder {
1127         return dec.compileSingle(remoteId, ut)
1128     }
1129     var wireStruct *structType
1130     // Builtin types can come from global pool; the rest
1131     // Also we know we're decoding a struct now, so the
1132     if t, ok := builtinIdToType[remoteId]; ok {
1133         wireStruct, _ = t.(*structType)
1134     } else {
1135         wire := dec.wireType[remoteId]

```

```

1128         if wire == nil {
1129             error_(errBadType)
1130         }
1131         wireStruct = wire.StructT
1132     }
1133     if wireStruct == nil {
1134         errorf("type mismatch in decoder: want struc
1135     }
1136     engine = new(decEngine)
1137     engine.instr = make([]decInstr, len(wireStruct.Field
1138     seen := make(map[reflect.Type]*decOp)
1139     // Loop over the fields of the wire type.
1140     for fieldnum := 0; fieldnum < len(wireStruct.Field);
1141         wireField := wireStruct.Field[fieldnum]
1142         if wireField.Name == "" {
1143             errorf("empty name for remote field
1144         }
1145         ovfl := overflow(wireField.Name)
1146         // Find the field of the local type with the
1147         localField, present := srt.FieldByName(wireF
1148         // TODO(r): anonymous names
1149         if !present || !isExported(wireField.Name) {
1150             op := dec.decIgnoreOpFor(wireField.I
1151             engine.instr[fieldnum] = decInstr{op
1152             continue
1153         }
1154         if !dec.compatibleType(localField.Type, wire
1155             errorf("wrong type (%s) for received
1156         }
1157         op, indir := dec.decOpFor(wireField.Id, loca
1158         engine.instr[fieldnum] = decInstr{*op, field
1159         engine.numInstr++
1160     }
1161     return
1162 }
1163
1164 // getDecEnginePtr returns the engine for the specified type
1165 func (dec *Decoder) getDecEnginePtr(remoteId typeId, ut *use
1166     rt := ut.user
1167     decoderMap, ok := dec.decoderCache[rt]
1168     if !ok {
1169         decoderMap = make(map[typeId]**decEngine)
1170         dec.decoderCache[rt] = decoderMap
1171     }
1172     if enginePtr, ok = decoderMap[remoteId]; !ok {
1173         // To handle recursive types, mark this engi
1174         enginePtr = new(*decEngine)
1175         decoderMap[remoteId] = enginePtr
1176         *enginePtr, err = dec.compileDec(remoteId, u

```

```

1177         if err != nil {
1178             delete(decoderMap, remoteId)
1179         }
1180     }
1181     return
1182 }
1183
1184 // emptyStruct is the type we compile into when ignoring a s
1185 type emptyStruct struct{}
1186
1187 var emptyStructType = reflect.TypeOf(emptyStruct{})
1188
1189 // getDecEnginePtr returns the engine for the specified type
1190 func (dec *Decoder) getIgnoreEnginePtr(wireId typeId) (engin
1191     var ok bool
1192     if enginePtr, ok = dec.ignorerCache[wireId]; !ok {
1193         // To handle recursive types, mark this engi
1194         enginePtr = new(*decEngine)
1195         dec.ignorerCache[wireId] = enginePtr
1196         wire := dec.wireType[wireId]
1197         if wire != nil && wire.StructT != nil {
1198             *enginePtr, err = dec.compileDec(wir
1199         } else {
1200             *enginePtr, err = dec.compileIgnoreS
1201         }
1202         if err != nil {
1203             delete(dec.ignorerCache, wireId)
1204         }
1205     }
1206     return
1207 }
1208
1209 // decodeValue decodes the data stream representing a value
1210 func (dec *Decoder) decodeValue(wireId typeId, val reflect.v
1211     defer catchError(&dec.err)
1212     // If the value is nil, it means we should just igno
1213     if !val.IsValid() {
1214         dec.decodeIgnoredValue(wireId)
1215         return
1216     }
1217     // Dereference down to the underlying type.
1218     ut := userType(val.Type())
1219     base := ut.base
1220     var enginePtr **decEngine
1221     enginePtr, dec.err = dec.getDecEnginePtr(wireId, ut)
1222     if dec.err != nil {
1223         return
1224     }
1225     engine := *enginePtr

```

```

1226         if st := base; st.Kind() == reflect.Struct && !ut.is
1227             if engine.numInstr == 0 && st.NumField() > 0
1228                 name := base.Name()
1229                 errorf("type mismatch: no fields mat
1230             }
1231             dec.decodeStruct(engine, ut, uintptr(unsafeA
1232     } else {
1233         dec.decodeSingle(engine, ut, uintptr(unsafeA
1234     }
1235 }
1236
1237 // decodeIgnoredValue decodes the data stream representing a
1238 func (dec *Decoder) decodeIgnoredValue(wireId typeId) {
1239     var enginePtr **decEngine
1240     enginePtr, dec.err = dec.getIgnoreEnginePtr(wireId)
1241     if dec.err != nil {
1242         return
1243     }
1244     wire := dec.wireType[wireId]
1245     if wire != nil && wire.StructT != nil {
1246         dec.ignoreStruct(*enginePtr)
1247     } else {
1248         dec.ignoreSingle(*enginePtr)
1249     }
1250 }
1251
1252 func init() {
1253     var iop, uop decOp
1254     switch reflect.TypeOf(int(0)).Bits() {
1255     case 32:
1256         iop = decInt32
1257         uop = decUInt32
1258     case 64:
1259         iop = decInt64
1260         uop = decUInt64
1261     default:
1262         panic("gob: unknown size of int/uint")
1263     }
1264     decOpTable[reflect.Int] = iop
1265     decOpTable[reflect.Uint] = uop
1266
1267     // Finally uintptr
1268     switch reflect.TypeOf(uintptr(0)).Bits() {
1269     case 32:
1270         uop = decUInt32
1271     case 64:
1272         uop = decUInt64
1273     default:
1274         panic("gob: unknown size of uintptr")
1275     }

```

```

1276         decOpTable[reflect.Uintptr] = uop
1277     }
1278
1279     // Gob assumes it can call UnsafeAddr on any Value
1280     // in order to get a pointer it can copy data from.
1281     // Values that have just been created and do not point
1282     // into existing structs or slices cannot be addressed,
1283     // so simulate it by returning a pointer to a copy.
1284     // Each call allocates once.
1285     func unsafeAddr(v reflect.Value) uintptr {
1286         if v.CanAddr() {
1287             return v.UnsafeAddr()
1288         }
1289         x := reflect.New(v.Type()).Elem()
1290         x.Set(v)
1291         return x.UnsafeAddr()
1292     }
1293
1294     // Gob depends on being able to take the address
1295     // of zeroed Values it creates, so use this wrapper instead
1296     // of the standard reflect.Zero.
1297     // Each call allocates once.
1298     func allocValue(t reflect.Type) reflect.Value {
1299         return reflect.New(t).Elem()
1300     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/gob/decoder.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package gob
6
7 import (
8     "bufio"
9     "bytes"
10    "errors"
11    "io"
12    "reflect"
13    "sync"
14 )
15
16 // A Decoder manages the receipt of type and data informatio
17 // remote side of a connection.
18 type Decoder struct {
19     mutex      sync.Mutex
20     r          io.Reader
21     buf        bytes.Buffer
22     wireType   map[typeId]*wireType
23     decoderCache map[reflect.Type]map[typeId]**decEngine
24     ignorerCache map[typeId]**decEngine
25     freeList   *decoderState
26     countBuf   []byte
27     tmp        []byte
28     err        error
29 }
30
31 // NewDecoder returns a new decoder that reads from the io.R
32 // If r does not also implement io.ByteReader, it will be wr
33 // bufio.Reader.
34 func NewDecoder(r io.Reader) *Decoder {
35     dec := new(Decoder)
36     // We use the ability to read bytes as a plausible s
37     if _, ok := r.(io.ByteReader); !ok {
38         r = bufio.NewReader(r)
39     }
40     dec.r = r
41     dec.wireType = make(map[typeId]*wireType)
```

```

42         dec.decoderCache = make(map[reflect.Type]map[typeId]
43         dec.ignorerCache = make(map[typeId]**decEngine)
44         dec.countBuf = make([]byte, 9) // counts may be uint
45
46         return dec
47     }
48
49     // recvType loads the definition of a type.
50     func (dec *Decoder) recvType(id typeId) {
51         // Have we already seen this type? That's an error
52         if id < firstUserId || dec.wireType[id] != nil {
53             dec.err = errors.New("gob: duplicate type re
54             return
55         }
56
57         // Type:
58         wire := new(wireType)
59         dec.decodeValue(tWireType, reflect.ValueOf(wire))
60         if dec.err != nil {
61             return
62         }
63         // Remember we've seen this type.
64         dec.wireType[id] = wire
65     }
66
67     var errBadCount = errors.New("invalid message length")
68
69     // recvMessage reads the next count-delimited item from the
70     // of Encoder.writeMessage. It returns false on EOF or other
71     func (dec *Decoder) recvMessage() bool {
72         // Read a count.
73         nbytes, _, err := decodeUintReader(dec.r, dec.countB
74         if err != nil {
75             dec.err = err
76             return false
77         }
78         // Upper limit of 1GB, allowing room to grow a littl
79         // TODO: We might want more control over this limit.
80         if nbytes >= 1<<30 {
81             dec.err = errBadCount
82             return false
83         }
84         dec.readMessage(int(nbytes))
85         return dec.err == nil
86     }
87
88     // readMessage reads the next nbytes bytes from the input.
89     func (dec *Decoder) readMessage(nbytes int) {
90         // Allocate the buffer.
91         if cap(dec.tmp) < nbytes {

```

```

92         dec.tmp = make([]byte, nbytes+100) // room t
93     }
94     dec.tmp = dec.tmp[:nbytes]
95
96     // Read the data
97     _, dec.err = io.ReadFull(dec.r, dec.tmp)
98     if dec.err != nil {
99         if dec.err == io.EOF {
100             dec.err = io.ErrUnexpectedEOF
101         }
102         return
103     }
104     dec.buf.Write(dec.tmp)
105 }
106
107 // toInt turns an encoded uint64 into an int, according to t
108 func toInt(x uint64) int64 {
109     i := int64(x >> 1)
110     if x&1 != 0 {
111         i = ^i
112     }
113     return i
114 }
115
116 func (dec *Decoder) nextInt() int64 {
117     n, _, err := decodeUintReader(&dec.buf, dec.countBuf)
118     if err != nil {
119         dec.err = err
120     }
121     return toInt(n)
122 }
123
124 func (dec *Decoder) nextUint() uint64 {
125     n, _, err := decodeUintReader(&dec.buf, dec.countBuf)
126     if err != nil {
127         dec.err = err
128     }
129     return n
130 }
131
132 // decodeTypeSequence parses:
133 // TypeSequence
134 //     (TypeDefinition DelimitedTypeDefinition*)?
135 // and returns the type id of the next value. It returns -1
136 // EOF. Upon return, the remainder of dec.buf is the value
137 // decoded. If this is an interface value, it can be ignore
138 // resetting that buffer.
139 func (dec *Decoder) decodeTypeSequence(isInterface bool) typ
140     for dec.err == nil {

```

```

141         if dec.buf.Len() == 0 {
142             if !dec.recvMessage() {
143                 break
144             }
145         }
146         // Receive a type id.
147         id := typeId(dec.nextInt())
148         if id >= 0 {
149             // Value follows.
150             return id
151         }
152         // Type definition for (-id) follows.
153         dec.recvType(-id)
154         // When decoding an interface, after a type
155         // DelimitedValue still in the buffer. Skip
156         // (Alternatively, the buffer is empty and t
157         // will be absorbed by recvMessage.)
158         if dec.buf.Len() > 0 {
159             if !isInterface {
160                 dec.err = errors.New("extra
161                 break
162             }
163             dec.nextUint()
164         }
165     }
166     return -1
167 }
168
169 // Decode reads the next value from the connection and store
170 // it in the data represented by the empty interface value.
171 // If e is nil, the value will be discarded. Otherwise,
172 // the value underlying e must be a pointer to the
173 // correct type for the next data item received.
174 func (dec *Decoder) Decode(e interface{}) error {
175     if e == nil {
176         return dec.DecodeValue(reflect.Value{})
177     }
178     value := reflect.ValueOf(e)
179     // If e represents a value as opposed to a pointer,
180     // get back to the caller. Make sure it's a pointer
181     if value.Type().Kind() != reflect.Ptr {
182         dec.err = errors.New("gob: attempt to decode
183         return dec.err
184     }
185     return dec.DecodeValue(value)
186 }
187
188 // DecodeValue reads the next value from the connection.
189 // If v is the zero reflect.Value (v.Kind() == Invalid), Dec

```

```

190 // Otherwise, it stores the value into v. In that case, v must
191 // a non-nil pointer to data or be an assignable reflect.Value
192 func (dec *Decoder) DecodeValue(v reflect.Value) error {
193     if v.IsValid() {
194         if v.Kind() == reflect.Ptr && !v.IsNil() {
195             // That's okay, we'll store through
196         } else if !v.CanSet() {
197             return errors.New("gob: DecodeValue
198         }
199     }
200     // Make sure we're single-threaded through here.
201     dec.mutex.Lock()
202     defer dec.mutex.Unlock()
203
204     dec.buf.Reset() // In case data lingers from previous
205     dec.err = nil
206     id := dec.decodeTypeSequence(false)
207     if dec.err == nil {
208         dec.decodeValue(id, v)
209     }
210     return dec.err
211 }
212
213 // If debug.go is compiled into the program, debugFunc prints
214 // representation of the gob data read from r by calling the
215 // Otherwise it is nil.
216 var debugFunc func(io.Reader)

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/gob/doc.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6 Package gob manages streams of gobs - binary values exchange
7 Encoder (transmitter) and a Decoder (receiver). A typical u
8 arguments and results of remote procedure calls (RPCs) such
9 package "rpc".
10
11 A stream of gobs is self-describing. Each data item in the
12 a specification of its type, expressed in terms of a small s
13 types. Pointers are not transmitted, but the things they po
14 transmitted; that is, the values are flattened. Recursive t
15 recursive values (data with cycles) are problematic. This m
16
17 To use gobs, create an Encoder and present it with a series
18 values or addresses that can be dereferenced to values. The
19 all type information is sent before it is needed. At the re
20 Decoder retrieves values from the encoded stream and unpacks
21 variables.
22
23 The source and destination values/types need not correspond
24 fields (identified by name) that are in the source but absen
25 variable will be ignored. Fields that are in the receiving
26 from the transmitted type or value will be ignored in the de
27 with the same name is present in both, their types must be c
28 receiver and transmitter will do all necessary indirection a
29 convert between gobs and actual Go values. For instance, a
30 schematically,
31
32     struct { A, B int }
33
34 can be sent from or received into any of these Go types:
35
36     struct { A, B int }      // the same
37     *struct { A, B int }    // extra indirection of the
38     struct { *A, **B int }  // extra indirection of the
39     struct { A, B int64 }   // different concrete value
40
41 It may also be received into any of these:
```

```

42
43     struct { A, B int }      // the same
44     struct { B, A int }      // ordering doesn't matter;
45     struct { A, B, C int }   // extra field (C) ignored
46     struct { B int }         // missing field (A) ignored
47     struct { B, C int }      // missing field (A) ignored
48

```

49 Attempting to receive into these types will draw a decode error

```

50
51     struct { A int; B uint } // change of signedness
52     struct { A int; B float } // change of type for B
53     struct { }               // no field names in struct
54     struct { C, D int }      // no field names in struct
55

```

56 Integers are transmitted two ways: arbitrary precision signed and
57 arbitrary precision unsigned integers. There is no int8, in
58 discrimination in the gob format; there are only signed and
59 described below, the transmitter sends the value in a variable
60 the receiver accepts the value and stores it in the destination
61 Floating-point numbers are always sent using IEEE-754 64-bit
62 below).

63
64 Signed integers may be received into any signed integer variable
65 unsigned integers may be received into any unsigned integer
66 point values may be received into any floating point variable
67 the destination variable must be able to represent the value
68 operation will fail.

69
70 Structs, arrays and slices are also supported. Strings and
71 supported with a special, efficient representation (see below)
72 decoded, if the existing slice has capacity the slice will be
73 if not, a new array is allocated. Regardless, the length of
74 reports the number of elements decoded.

75
76 Functions and channels cannot be sent in a gob. Attempting
77 to encode a value that contains one will fail.

78
79 The rest of this comment documents the encoding, details that
80 for most users. Details are presented bottom-up.

81
82 An unsigned integer is sent one of two ways. If it is less
83 as a byte with that value. Otherwise it is sent as a minimal
84 (high byte first) byte stream holding the value, preceded by
85 byte count, negated. Thus 0 is transmitted as (00), 7 is transmitted
86 256 is transmitted as (FE 01 00).

87
88 A boolean is encoded within an unsigned integer: 0 for false

89
90 A signed integer, i, is encoded within an unsigned integer,
91 upward contain the value; bit 0 says whether they should be

```

92 receipt. The encode algorithm looks like this:
93
94     uint u;
95     if i < 0 {
96         u = (^i << 1) | 1      // complement i, bit
97     } else {
98         u = (i << 1)         // do not complement i, bit
99     }
100     encodeUnsigned(u)
101

```

102 The low bit is therefore analogous to a sign bit, but making
103 instead guarantees that the largest negative integer is not
104 example, $-129 = \wedge 128 = (\wedge 256 \gg 1)$ encodes as (FE 01 01).

105
106 Floating-point numbers are always sent as a representation o
107 That value is converted to a uint64 using math.Float64bits.
108 byte-reversed and sent as a regular unsigned integer. The b
109 exponent and high-precision part of the mantissa go first.
110 often zero, this can save encoding bytes. For instance, 17.
111 three bytes (FE 31 40).

112
113 Strings and slices of bytes are sent as an unsigned count fo
114 uninterpreted bytes of the value.

115
116 All other slices and arrays are sent as an unsigned count fo
117 elements using the standard gob encoding for their type, rec

118
119 Maps are sent as an unsigned count followed by that man key,
120 pairs. Empty but non-nil maps are sent, so if the sender has
121 a map, the receiver will allocate a map even no elements are
122 transmitted.

123
124 Structs are sent as a sequence of (field number, field value
125 value is sent using the standard gob encoding for its type,
126 field has the zero value for its type, it is omitted from th
127 field number is defined by the type of the encoded struct: t
128 encoded type is field 0, the second is field 1, etc. When e
129 field numbers are delta encoded for efficiency and the field
130 order of increasing field number; the deltas are therefore u
131 initialization for the delta encoding sets the field number
132 integer field 0 with value 7 is transmitted as unsigned delt
133 = 7 or (01 07). Finally, after all the fields have been sen
134 denotes the end of the struct. That mark is a delta=0 value
135 representation (00).

136
137 Interface types are not checked for compatibility; all inter
138 treated, for transmission, as members of a single "interface
139 int or []byte - in effect they're all treated as interface{}
140 are transmitted as a string identifying the concrete type be

141 that must be pre-defined by calling Register), followed by a
142 length of the following data (so the value can be skipped if
143 stored), followed by the usual encoding of concrete (dynamic
144 the interface value. (A nil interface value is identified b
145 and transmits no value.) Upon receipt, the decoder verifies
146 concrete item satisfies the interface of the receiving varia
147

148 The representation of types is described below. When a type
149 connection between an Encoder and Decoder, it is assigned a
150 id. When Encoder.Encode(v) is called, it makes sure there i
151 the type of v and all its elements and then it sends the pai
152 where typeid is the type id of the encoded type of v and enc
153 encoding of the value v.

154

155 To define a type, the encoder chooses an unused, positive ty
156 pair (-type id, encoded-type) where encoded-type is the gob
157 description, constructed from these types:

158

```
159     type wireType struct {
160         ArrayT *ArrayType
161         SliceT *SliceType
162         StructT *StructType
163         MapT *MapType
164     }
165     type arrayType struct {
166         CommonType
167         Elem typeId
168         Len int
169     }
170     type CommonType struct {
171         Name string // the name of the struct type
172         Id int // the id of the type, repeated s
173     }
174     type sliceType struct {
175         CommonType
176         Elem typeId
177     }
178     type structType struct {
179         CommonType
180         Field []*fieldType // the fields of the stru
181     }
182     type fieldType struct {
183         Name string // the name of the field.
184         Id int // the type id of the field, whi
185     }
186     type mapType struct {
187         CommonType
188         Key typeId
189         Elem typeId
```

```

190     }
191
192 If there are nested type ids, the types for all inner type i
193 before the top-level type id is used to describe an encoded-
194
195 For simplicity in setup, the connection is defined to unders
196 priori, as well as the basic gob types int, uint, etc. Thei
197
198     bool        1
199     int         2
200     uint        3
201     float       4
202     []byte      5
203     string      6
204     complex     7
205     interface   8
206     // gap for reserved ids.
207     wireType    16
208     ArrayType   17
209     CommonType  18
210     SliceType   19
211     StructType  20
212     FieldType   21
213     // 22 is slice of fieldType.
214     MapType     23
215
216 Finally, each message created by a call to Encode is precede
217 unsigned integer count of the number of bytes remaining in t
218 the initial type name, interface values are wrapped the same
219 interface value acts like a recursive invocation of Encode.
220
221 In summary, a gob stream looks like
222
223     (byteCount (-type id, encoding of a wireType)* (type
224
225 where * signifies zero or more repetitions and the type id c
226 be predefined or be defined before the value in the stream.
227
228 See "Gobs of data" for a design discussion of the gob wire f
229 http://golang.org/doc/articles/gobs\_of\_data.html
230 */
231 package gob
232
233 /*
234 Grammar:
235
236 Tokens starting with a lower case letter are terminals; int(
237 and uint(n) represent the signed/unsigned encodings of the v
238
239 GobStream:

```

```

240         DelimitedMessage*
241 DelimitedMessage:
242         uint(lengthOfMessage) Message
243 Message:
244         TypeSequence TypedValue
245 TypeSequence
246         (TypeDefinition DelimitedTypeDefinition*)?
247 DelimitedTypeDefinition:
248         uint(lengthOfTypeDefinition) TypeDefinition
249 TypedValue:
250         int(typeId) Value
251 TypeDefinition:
252         int(-typeId) encodingOfWireType
253 Value:
254         SingletonValue | StructValue
255 SingletonValue:
256         uint(0) FieldValue
257 FieldValue:
258         builtinValue | ArrayValue | MapValue | SliceValue |
259 InterfaceValue:
260         NilInterfaceValue | NonNilInterfaceValue
261 NilInterfaceValue:
262         uint(0)
263 NonNilInterfaceValue:
264         ConcreteTypeName TypeSequence InterfaceContents
265 ConcreteTypeName:
266         uint(lengthOfName) [already read=n] name
267 InterfaceContents:
268         int(concreteTypeId) DelimitedValue
269 DelimitedValue:
270         uint(length) Value
271 ArrayValue:
272         uint(n) FieldValue*n [n elements]
273 MapValue:
274         uint(n) (FieldValue FieldValue)*n [n (key, value) p
275 SliceValue:
276         uint(n) FieldValue*n [n elements]
277 StructValue:
278         (uint(fieldDelta) FieldValue)*
279 */
280
281 /*
282 For implementers and the curious, here is an encoded example
283     type Point struct {X, Y int}
284 and the value
285     p := Point{22, 33}
286 the bytes transmitted that encode p will be:
287     1f ff 81 03 01 01 05 50 6f 69 6e 74 01 ff 82 00
288     01 02 01 01 58 01 04 00 01 01 59 01 04 00 00 00

```

```

289         07 ff 82 01 2c 01 42 00
290 They are determined as follows.
291
292 Since this is the first transmission of type Point, the type
293 for Point itself must be sent before the value. This is the
294 we've sent on this Encoder, so it has type id 65 (0 through
295 reserved).
296
297         1f          // This item (a type descriptor) is 31 bytes
298         ff 81      // The negative of the id for the type we're
299                  // This is one byte (indicated by FF = -1) f
300                  // ^-65<<1 | 1. The low 1 bit signals to co
301                  // rest upon receipt.
302
303         // Now we send a type descriptor, which is itself a
304         // The type of wireType itself is known (it's built
305         // all its components), so we just need to send a *v
306         // that represents type "Point".
307         // Here starts the encoding of that value.
308         // Set the field number implicitly to -1; this is do
309         // of every struct, including nested structs.
310         03          // Add 3 to field number; now 2 (wireType.st
311                  // structType starts with an embedded Common
312                  // as a regular structure here too.
313         01          // add 1 to field number (now 0); start of e
314         01          // add 1 to field number (now 0, the name of
315         05          // string is (unsigned) 5 bytes long
316         50 6f 69 6e 74 // wireType.structType.CommonType.na
317         01          // add 1 to field number (now 1, the id of t
318         ff 82      // wireType.structType.CommonType._id = 65
319         00          // end of embedded wireType.structType.Commo
320         01          // add 1 to field number (now 1, the field a
321         02          // There are two fields in the type (len(str
322         01          // Start of first field structure; add 1 to
323         01          // 1 byte
324         58          // structType.field[0].name = "X"
325         01          // Add 1 to get field number 1: field[0].id
326         04          // structType.field[0].typeId is 2 (signed i
327         00          // End of structType.field[0]; start structT
328         01          // Add 1 to get field number 0: field[1].nam
329         01          // 1 byte
330         59          // structType.field[1].name = "Y"
331         01          // Add 1 to get field number 1: field[0].id
332         04          // struct.Type.field[1].typeId is 2 (signed
333         00          // End of structType.field[1]; end of struct
334         00          // end of wireType.structType structure
335         00          // end of wireType structure
336
337 Now we can send the Point value. Again the field number res

```

```

338
339     07      // this value is 7 bytes long
340     ff 82   // the type number, 65 (1 byte (-FF) followe
341     01      // add one to field number, yielding field 0
342     2c      // encoding of signed "22" (0x22 = 44 = 22<<
343     01      // add one to field number, yielding field 1
344     42      // encoding of signed "33" (0x42 = 66 = 33<<
345     00      // end of structure
346
347 The type encoding is long and fairly intricate but we send i
348 If p is transmitted a second time, the type is already known
349 output will be just:
350
351     07 ff 82 01 2c 01 42 00
352
353 A single non-struct value at top level is transmitted like a
354 delta tag 0. For instance, a signed integer with value 3 pr
355 the argument to Encode will emit:
356
357     03 04 00 06
358
359 Which represents:
360
361     03      // this value is 3 bytes long
362     04      // the type number, 2, represents an integer
363     00      // tag delta 0
364     06      // value 3
365
366 */

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/gob/encode.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package gob
6
7 import (
8     "bytes"
9     "math"
10    "reflect"
11    "unsafe"
12 )
13
14 const uint64Size = int(unsafe.Sizeof(uint64(0)))
15
16 // encoderState is the global execution state of an instance
17 // Field numbers are delta encoded and always increase. The
18 // number is initialized to -1 so 0 comes out as delta(1). A
19 // 0 terminates the structure.
20 type encoderState struct {
21     enc      *Encoder
22     b        *bytes.Buffer
23     sendZero bool           // encoding an array e
24     fieldnum int           // the last field numb
25     buf      [1 + uint64Size]byte // buffer used by the
26     next     *encoderState       // for free list
27 }
28
29 func (enc *Encoder) newEncoderState(b *bytes.Buffer) *encoderState {
30     e := enc.freeList
31     if e == nil {
32         e = new(encoderState)
33         e.enc = enc
34     } else {
35         enc.freeList = e.next
36     }
37     e.sendZero = false
38     e.fieldnum = 0
39     e.b = b
40     return e
41 }
```

```

42
43 func (enc *Encoder) freeEncoderState(e *encoderState) {
44     e.next = enc.freeList
45     enc.freeList = e
46 }
47
48 // Unsigned integers have a two-state encoding. If the numb
49 // than 128 (0 through 0x7F), its value is written directly.
50 // Otherwise the value is written in big-endian byte order p
51 // by the byte length, negated.
52
53 // encodeUint writes an encoded unsigned integer to state.b.
54 func (state *encoderState) encodeUint(x uint64) {
55     if x <= 0x7F {
56         err := state.b.WriteByte(uint8(x))
57         if err != nil {
58             error_(err)
59         }
60         return
61     }
62     i := uint64Size
63     for x > 0 {
64         state.buf[i] = uint8(x)
65         x >>= 8
66         i--
67     }
68     state.buf[i] = uint8(i - uint64Size) // = loop count
69     _, err := state.b.Write(state.buf[i : uint64Size+1])
70     if err != nil {
71         error_(err)
72     }
73 }
74
75 // encodeInt writes an encoded signed integer to state.w.
76 // The low bit of the encoding says whether to bit complemen
77 // uint to recover the int.
78 func (state *encoderState) encodeInt(i int64) {
79     var x uint64
80     if i < 0 {
81         x = uint64(^i<<1) | 1
82     } else {
83         x = uint64(i << 1)
84     }
85     state.encodeUint(uint64(x))
86 }
87
88 // encOp is the signature of an encoding operator for a give
89 type encOp func(i *encInstr, state *encoderState, p unsafe.P
90
91 // The 'instructions' of the encoding machine

```

```

92 type encInstr struct {
93     op      encOp
94     field  int      // field number
95     indir  int      // how many pointer indirections to r
96     offset uintptr // offset in the structure of the fie
97 }
98
99 // update emits a field number and updates the state to reco
100 // If the instruction pointer is nil, it does nothing
101 func (state *encoderState) update(instr *encInstr) {
102     if instr != nil {
103         state.encodeUint(uint64(instr.field - state.
104             state.fieldnum = instr.field
105     }
106 }
107
108 // Each encoder for a composite is responsible for handling
109 // indirections associated with the elements of the data str
110 // If any pointer so reached is nil, no bytes are written.
111 // data item is zero, no bytes are written. Single values -
112 // strings etc. - are indirected before calling their encode
113 // Otherwise, the output (for a scalar) is the field number,
114 // encoded integer, followed by the field data in its approp
115 // format.
116
117 // encIndirect dereferences p indir times and returns the re
118 func encIndirect(p unsafe.Pointer, indir int) unsafe.Pointer
119     for ; indir > 0; indir-- {
120         p = *(*unsafe.Pointer)(p)
121         if p == nil {
122             return unsafe.Pointer(nil)
123         }
124     }
125     return p
126 }
127
128 // encBool encodes the bool with address p as an unsigned 0
129 func encBool(i *encInstr, state *encoderState, p unsafe.Poin
130     b := *(*bool)(p)
131     if b || state.sendZero {
132         state.update(i)
133         if b {
134             state.encodeUint(1)
135         } else {
136             state.encodeUint(0)
137         }
138     }
139 }
140

```

```

141 // encInt encodes the int with address p.
142 func encInt(i *encInstr, state *encoderState, p unsafe.Point
143     v := int64>(*int)(p))
144     if v != 0 || state.sendZero {
145         state.update(i)
146         state.encodeInt(v)
147     }
148 }
149
150 // encUint encodes the uint with address p.
151 func encUint(i *encInstr, state *encoderState, p unsafe.Poin
152     v := uint64>(*uint)(p))
153     if v != 0 || state.sendZero {
154         state.update(i)
155         state.encodeUint(v)
156     }
157 }
158
159 // encInt8 encodes the int8 with address p.
160 func encInt8(i *encInstr, state *encoderState, p unsafe.Poin
161     v := int64>(*int8)(p))
162     if v != 0 || state.sendZero {
163         state.update(i)
164         state.encodeInt(v)
165     }
166 }
167
168 // encUint8 encodes the uint8 with address p.
169 func encUint8(i *encInstr, state *encoderState, p unsafe.Poi
170     v := uint64>(*uint8)(p))
171     if v != 0 || state.sendZero {
172         state.update(i)
173         state.encodeUint(v)
174     }
175 }
176
177 // encInt16 encodes the int16 with address p.
178 func encInt16(i *encInstr, state *encoderState, p unsafe.Poi
179     v := int64>(*int16)(p))
180     if v != 0 || state.sendZero {
181         state.update(i)
182         state.encodeInt(v)
183     }
184 }
185
186 // encUint16 encodes the uint16 with address p.
187 func encUint16(i *encInstr, state *encoderState, p unsafe.Po
188     v := uint64>(*uint16)(p))
189     if v != 0 || state.sendZero {

```

```

190             state.update(i)
191             state.encodeUint(v)
192         }
193     }
194
195     // encInt32 encodes the int32 with address p.
196     func encInt32(i *encInstr, state *encoderState, p unsafe.Poi
197         v := int64>(*int32)(p))
198         if v != 0 || state.sendZero {
199             state.update(i)
200             state.encodeInt(v)
201         }
202     }
203
204     // encUint encodes the uint32 with address p.
205     func encUint32(i *encInstr, state *encoderState, p unsafe.Po
206         v := uint64>(*uint32)(p))
207         if v != 0 || state.sendZero {
208             state.update(i)
209             state.encodeUint(v)
210         }
211     }
212
213     // encInt64 encodes the int64 with address p.
214     func encInt64(i *encInstr, state *encoderState, p unsafe.Poi
215         v := *(*int64)(p)
216         if v != 0 || state.sendZero {
217             state.update(i)
218             state.encodeInt(v)
219         }
220     }
221
222     // encInt64 encodes the uint64 with address p.
223     func encUint64(i *encInstr, state *encoderState, p unsafe.Po
224         v := *(*uint64)(p)
225         if v != 0 || state.sendZero {
226             state.update(i)
227             state.encodeUint(v)
228         }
229     }
230
231     // encUintptr encodes the uintptr with address p.
232     func encUintptr(i *encInstr, state *encoderState, p unsafe.P
233         v := uint64>(*uintptr)(p))
234         if v != 0 || state.sendZero {
235             state.update(i)
236             state.encodeUint(v)
237         }
238     }
239

```

```

240 // floatBits returns a uint64 holding the bits of a floating
241 // Floating-point numbers are transmitted as uint64s holding
242 // of the underlying representation. They are sent byte-rev
243 // the exponent end coming out first, so integer floating po
244 // (for example) transmit more compactly. This routine does
245 // swizzling.
246 func floatBits(f float64) uint64 {
247     u := math.Float64bits(f)
248     var v uint64
249     for i := 0; i < 8; i++ {
250         v <<= 8
251         v |= u & 0xFF
252         u >>= 8
253     }
254     return v
255 }
256
257 // encFloat32 encodes the float32 with address p.
258 func encFloat32(i *encInstr, state *encoderState, p unsafe.P
259     f :=>(*float32)(p)
260     if f != 0 || state.sendZero {
261         v := floatBits(float64(f))
262         state.update(i)
263         state.encodeUint(v)
264     }
265 }
266
267 // encFloat64 encodes the float64 with address p.
268 func encFloat64(i *encInstr, state *encoderState, p unsafe.P
269     f :=>(*float64)(p)
270     if f != 0 || state.sendZero {
271         state.update(i)
272         v := floatBits(f)
273         state.encodeUint(v)
274     }
275 }
276
277 // encComplex64 encodes the complex64 with address p.
278 // Complex numbers are just a pair of floating-point numbers
279 func encComplex64(i *encInstr, state *encoderState, p unsafe
280     c :=>(*complex64)(p)
281     if c != 0+0i || state.sendZero {
282         rpart := floatBits(float64(real(c)))
283         ipart := floatBits(float64(imag(c)))
284         state.update(i)
285         state.encodeUint(rpart)
286         state.encodeUint(ipart)
287     }
288 }

```

```

289
290 // encComplex128 encodes the complex128 with address p.
291 func encComplex128(i *encInstr, state *encoderState, p unsafe
292     c := *(*complex128)(p)
293     if c != 0+0i || state.sendZero {
294         rpart := floatBits(real(c))
295         ipart := floatBits(imag(c))
296         state.update(i)
297         state.encodeUint(rpart)
298         state.encodeUint(ipart)
299     }
300 }
301
302 // encUint8Array encodes the byte slice whose header has add
303 // Byte arrays are encoded as an unsigned count followed by
304 func encUint8Array(i *encInstr, state *encoderState, p unsafe
305     b := *(*[]byte)(p)
306     if len(b) > 0 || state.sendZero {
307         state.update(i)
308         state.encodeUint(uint64(len(b)))
309         state.b.Write(b)
310     }
311 }
312
313 // encString encodes the string whose header has address p.
314 // Strings are encoded as an unsigned count followed by the
315 func encString(i *encInstr, state *encoderState, p unsafe.Pointer
316     s := *(*string)(p)
317     if len(s) > 0 || state.sendZero {
318         state.update(i)
319         state.encodeUint(uint64(len(s)))
320         state.b.WriteString(s)
321     }
322 }
323
324 // encStructTerminator encodes the end of an encoded struct
325 // as delta field number of 0.
326 func encStructTerminator(i *encInstr, state *encoderState, p
327     state.encodeUint(0)
328 }
329
330 // Execution engine
331
332 // encEngine an array of instructions indexed by field number
333 // data, typically a struct. It is executed top to bottom,
334 type encEngine struct {
335     instr []encInstr
336 }
337

```

```

338 const singletonField = 0
339
340 // encodeSingle encodes a single top-level non-struct value.
341 func (enc *Encoder) encodeSingle(b *bytes.Buffer, engine *en
342     state := enc.newEncoderState(b)
343     state.fieldnum = singletonField
344     // There is no surrounding struct to frame the trans
345     // generate data even if the item is zero. To do th
346     state.sendZero = true
347     instr := &engine.instr[singletonField]
348     p := unsafe.Pointer(basep) // offset will be zero
349     if instr.indir > 0 {
350         if p = encIndirect(p, instr.indir); p == nil
351             return
352     }
353 }
354 instr.op(instr, state, p)
355 enc.freeEncoderState(state)
356 }
357
358 // encodeStruct encodes a single struct value.
359 func (enc *Encoder) encodeStruct(b *bytes.Buffer, engine *en
360     state := enc.newEncoderState(b)
361     state.fieldnum = -1
362     for i := 0; i < len(engine.instr); i++ {
363         instr := &engine.instr[i]
364         p := unsafe.Pointer(basep + instr.offset)
365         if instr.indir > 0 {
366             if p = encIndirect(p, instr.indir);
367                 continue
368         }
369     }
370     instr.op(instr, state, p)
371 }
372 enc.freeEncoderState(state)
373 }
374
375 // encodeArray encodes the array whose 0th element is at p.
376 func (enc *Encoder) encodeArray(b *bytes.Buffer, p uintptr,
377     state := enc.newEncoderState(b)
378     state.fieldnum = -1
379     state.sendZero = true
380     state.encodeUint(uint64(length))
381     for i := 0; i < length; i++ {
382         elemp := p
383         up := unsafe.Pointer(elemp)
384         if elemIndir > 0 {
385             if up = encIndirect(up, elemIndir);
386                 errorf("encodeArray: nil ele
387         }

```

```

388             elemp = uintptr(up)
389         }
390         op(nil, state, unsafe.Pointer(elemp))
391         p += uintptr(elemWid)
392     }
393     enc.freeEncoderState(state)
394 }
395
396 // encodeReflectValue is a helper for maps. It encodes the v
397 func encodeReflectValue(state *encoderState, v reflect.Value
398     for i := 0; i < indir && v.IsValid(); i++ {
399         v = reflect.Indirect(v)
400     }
401     if !v.IsValid() {
402         errorf("encodeReflectValue: nil element")
403     }
404     op(nil, state, unsafe.Pointer(unsafeAddr(v)))
405 }
406
407 // encodeMap encodes a map as unsigned count followed by key
408 // Because map internals are not exposed, we must use reflec
409 // addresses.
410 func (enc *Encoder) encodeMap(b *bytes.Buffer, mv reflect.Va
411     state := enc.newEncoderState(b)
412     state.fieldnum = -1
413     state.sendZero = true
414     keys := mv.MapKeys()
415     state.encodeUint(uint64(len(keys)))
416     for _, key := range keys {
417         encodeReflectValue(state, key, keyOp, keyInd
418         encodeReflectValue(state, mv.MapIndex(key),
419     }
420     enc.freeEncoderState(state)
421 }
422
423 // encodeInterface encodes the interface value iv.
424 // To send an interface, we send a string identifying the co
425 // by the type identifier (which might require defining that
426 // by the concrete value. A nil value gets sent as the empt
427 // followed by no value.
428 func (enc *Encoder) encodeInterface(b *bytes.Buffer, iv refl
429     state := enc.newEncoderState(b)
430     state.fieldnum = -1
431     state.sendZero = true
432     if iv.IsNil() {
433         state.encodeUint(0)
434         return
435     }
436

```

```

437         ut := userType(iv.Elem().Type())
438         name, ok := concreteTypeToName[ut.base]
439         if !ok {
440             errorf("type not registered for interface: %
441         }
442         // Send the name.
443         state.encodeUint(uint64(len(name)))
444         _, err := state.b.WriteString(name)
445         if err != nil {
446             error_(err)
447         }
448         // Define the type id if necessary.
449         enc.sendTypeDescriptor(enc.writer(), state, ut)
450         // Send the type id.
451         enc.sendTypeId(state, ut)
452         // Encode the value into a new buffer. Any nested t
453         // should be written to b, before the encoded value.
454         enc.pushWriter(b)
455         data := new(bytes.Buffer)
456         data.Write(spaceForLength)
457         enc.encode(data, iv.Elem(), ut)
458         if enc.err != nil {
459             error_(enc.err)
460         }
461         enc.popWriter()
462         enc.writeMessage(b, data)
463         if enc.err != nil {
464             error_(err)
465         }
466         enc.freeEncoderState(state)
467     }
468
469     // isZero returns whether the value is the zero of its type.
470     func isZero(val reflect.Value) bool {
471         switch val.Kind() {
472         case reflect.Array:
473             for i := 0; i < val.Len(); i++ {
474                 if !isZero(val.Index(i)) {
475                     return false
476                 }
477             }
478             return true
479         case reflect.Map, reflect.Slice, reflect.String:
480             return val.Len() == 0
481         case reflect.Bool:
482             return !val.Bool()
483         case reflect.Complex64, reflect.Complex128:
484             return val.Complex() == 0
485         case reflect.Chan, reflect.Func, reflect.Ptr:

```

```

486         return val.IsNil()
487     case reflect.Int, reflect.Int8, reflect.Int16, refle
488         return val.Int() == 0
489     case reflect.Float32, reflect.Float64:
490         return val.Float() == 0
491     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
492         return val.Uint() == 0
493     case reflect.Struct:
494         for i := 0; i < val.NumField(); i++ {
495             if !isZero(val.Field(i)) {
496                 return false
497             }
498         }
499         return true
500     }
501     panic("unknown type in isZero " + val.Type().String(
502 }
503
504 // encGobEncoder encodes a value that implements the GobEnco
505 // The data is sent as a byte array.
506 func (enc *Encoder) encodeGobEncoder(b *bytes.Buffer, v refl
507     // TODO: should we catch panics from the called meth
508     // We know it's a GobEncoder, so just call the metho
509     data, err := v.Interface().(GobEncoder).GobEncode()
510     if err != nil {
511         error_(err)
512     }
513     state := enc.newEncoderState(b)
514     state.fieldnum = -1
515     state.encodeUint(uint64(len(data)))
516     state.b.Write(data)
517     enc.freeEncoderState(state)
518 }
519
520 var encOpTable = [...]encOp{
521     reflect.Bool:      encBool,
522     reflect.Int:       encInt,
523     reflect.Int8:      encInt8,
524     reflect.Int16:     encInt16,
525     reflect.Int32:     encInt32,
526     reflect.Int64:     encInt64,
527     reflect.Uint:      encUint,
528     reflect.Uint8:     encUint8,
529     reflect.Uint16:    encUint16,
530     reflect.Uint32:    encUint32,
531     reflect.Uint64:    encUint64,
532     reflect.Uintptr:   encUintptr,
533     reflect.Float32:   encFloat32,
534     reflect.Float64:   encFloat64,
535     reflect.Complex64: encComplex64,

```

```

536         reflect.Complex128: encComplex128,
537         reflect.String:      encString,
538     }
539
540     // encOpFor returns (a pointer to) the encoding op for the b
541     // the indirection count to reach it.
542     func (enc *Encoder) encOpFor(rt reflect.Type, inProgress map
543         ut := userType(rt)
544         // If the type implements GobEncoder, we handle it w
545         if ut.isGobEncoder {
546             return enc.gobEncodeOpFor(ut)
547         }
548         // If this type is already in progress, it's a recur
549         // Return the pointer to the op we're already buildi
550         if opPtr := inProgress[rt]; opPtr != nil {
551             return opPtr, ut.indir
552         }
553         typ := ut.base
554         indir := ut.indir
555         k := typ.Kind()
556         var op encOp
557         if int(k) < len(encOpTable) {
558             op = encOpTable[k]
559         }
560         if op == nil {
561             inProgress[rt] = &op
562             // Special cases
563             switch t := typ; t.Kind() {
564             case reflect.Slice:
565                 if t.Elem().Kind() == reflect.Uint8
566                     op = encUint8Array
567                     break
568             }
569             // Slices have a header; we decode i
570             elemOp, indir := enc.encOpFor(t.Elem
571             op = func(i *encInstr, state *encode
572                 slice := (*reflect.SliceHead
573                 if !state.sendZero && slice.
574                     return
575                 }
576                 state.update(i)
577                 state.enc.encodeArray(state.
578             }
579             case reflect.Array:
580                 // True arrays have size in the type
581                 elemOp, indir := enc.encOpFor(t.Elem
582                 op = func(i *encInstr, state *encode
583                     state.update(i)
584                     state.enc.encodeArray(state.

```

```

585     }
586     case reflect.Map:
587         keyOp, keyIndir := enc.encOpFor(t.Key)
588         elemOp, elemIndir := enc.encOpFor(t.Elem)
589         op = func(i *encInstr, state *encode
590             // Maps cannot be accessed by
591             // that slices etc. can. We
592             // the iteration.
593             v := reflect.NewAt(t, unsafe
594             mv := reflect.Indirect(v)
595             // We send zero-length (but
596             // receiver might want to us
597             if !state.sendZero && mv.IsNil()
598                 return
599             }
600             state.update(i)
601             state.enc.encodeMap(state.b,
602         )
603     case reflect.Struct:
604         // Generate a closure that calls out
605         enc.getEncEngine(userType(typ))
606         info := mustGetTypeInfo(typ)
607         op = func(i *encInstr, state *encode
608             state.update(i)
609             // indirect through info to
610             state.enc.encodeStruct(state
611         )
612     case reflect.Interface:
613         op = func(i *encInstr, state *encode
614             // Interfaces transmit the n
615             // value they contain.
616             v := reflect.NewAt(t, unsafe
617             iv := reflect.Indirect(v)
618             if !state.sendZero && (!iv.IsNil()
619                 return
620             }
621             state.update(i)
622             state.enc.encodeInterface(st
623         )
624     }
625 }
626 if op == nil {
627     errorf("can't happen: encode type %s", rt)
628 }
629 return &op, indir
630 }
631
632 // gobEncodeOpFor returns the op for a type that is known to
633 // GobEncoder.

```

```

634 func (enc *Encoder) gobEncodeOpFor(ut *userInfo) (*encOp
635     rt := ut.user
636     if ut.encIndir == -1 {
637         rt = reflect.PtrTo(rt)
638     } else if ut.encIndir > 0 {
639         for i := int8(0); i < ut.encIndir; i++ {
640             rt = rt.Elem()
641         }
642     }
643     var op encOp
644     op = func(i *encInstr, state *encoderState, p unsafe
645         var v reflect.Value
646         if ut.encIndir == -1 {
647             // Need to climb up one level to tur
648             v = reflect.NewAt(rt, unsafe.Pointer
649         } else {
650             v = reflect.NewAt(rt, p).Elem()
651         }
652         if !state.sendZero && isZero(v) {
653             return
654         }
655         state.update(i)
656         state.enc.encodeGobEncoder(state.b, v)
657     }
658     return &op, int(ut.encIndir) // encIndir: op will ge
659 }
660
661 // compileEnc returns the engine to compile the type.
662 func (enc *Encoder) compileEnc(ut *userInfo) *encEngine
663     srt := ut.base
664     engine := new(encEngine)
665     seen := make(map[reflect.Type]*encOp)
666     rt := ut.base
667     if ut.isGobEncoder {
668         rt = ut.user
669     }
670     if !ut.isGobEncoder &&
671         srt.Kind() == reflect.Struct {
672         for fieldNum, wireFieldNum := 0, 0; fieldNum
673             f := srt.Field(fieldNum)
674             if !isExported(f.Name) {
675                 continue
676             }
677             op, indir := enc.encOpFor(f.Type, se
678             engine.instr = append(engine.instr,
679                 wireFieldNum++
680         }
681         if srt.NumField() > 0 && len(engine.instr) =
682             errorf("type %s has no exported fiel
683     }

```

```

684         engine.instr = append(engine.instr, encInstr
685     } else {
686         engine.instr = make([]encInstr, 1)
687         op, indir := enc.encOpFor(rt, seen)
688         engine.instr[0] = encInstr{*op, singletonFie
689     }
690     return engine
691 }
692
693 // getEncEngine returns the engine to compile the type.
694 // typeLock must be held (or we're in initialization and gua
695 func (enc *Encoder) getEncEngine(ut *userTypeInfo) *encEngin
696     info, err1 := getTypeInfo(ut)
697     if err1 != nil {
698         error_(err1)
699     }
700     if info.encoder == nil {
701         // mark this engine as underway before compi
702         info.encoder = new(encEngine)
703         info.encoder = enc.compileEnc(ut)
704     }
705     return info.encoder
706 }
707
708 // lockAndGetEncEngine is a function that locks and compiles
709 // This lets us hold the lock only while compiling, not when
710 func (enc *Encoder) lockAndGetEncEngine(ut *userTypeInfo) *e
711     typeLock.Lock()
712     defer typeLock.Unlock()
713     return enc.getEncEngine(ut)
714 }
715
716 func (enc *Encoder) encode(b *bytes.Buffer, value reflect.Va
717     defer catchError(&enc.err)
718     engine := enc.lockAndGetEncEngine(ut)
719     indir := ut.indir
720     if ut.isGobEncoder {
721         indir = int(ut.encIndir)
722     }
723     for i := 0; i < indir; i++ {
724         value = reflect.Indirect(value)
725     }
726     if !ut.isGobEncoder && value.Type().Kind() == reflec
727         enc.encodeStruct(b, engine, unsafeAddr(value)
728     } else {
729         enc.encodeSingle(b, engine, unsafeAddr(value)
730     }
731 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/gob/encoder.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package gob
6
7 import (
8     "bytes"
9     "errors"
10    "io"
11    "reflect"
12    "sync"
13 )
14
15 // An Encoder manages the transmission of type and data info
16 // other side of a connection.
17 type Encoder struct {
18     mutex      sync.Mutex           // each item must
19     w          []io.Writer          // where to send
20     sent       map[reflect.Type]typeId // which types we
21     countState *encoderState        // stage for writ
22     freeList   *encoderState        // list of free e
23     byteBuf   bytes.Buffer         // buffer for top
24     err       error
25 }
26
27 // Before we encode a message, we reserve space at the head
28 // buffer in which to encode its length. This means we can u
29 // buffer to assemble the message without another allocation
30 const maxLength = 9 // Maximum size of an encoded length.
31 var spaceForLength = make([]byte, maxLength)
32
33 // NewEncoder returns a new encoder that will transmit on th
34 func NewEncoder(w io.Writer) *Encoder {
35     enc := new(Encoder)
36     enc.w = []io.Writer{w}
37     enc.sent = make(map[reflect.Type]typeId)
38     enc.countState = enc.newEncoderState(new(bytes.Buffe
39     return enc
40 }
41
```

```

42 // writer() returns the innermost writer the encoder is using
43 func (enc *Encoder) writer() io.Writer {
44     return enc.w[len(enc.w)-1]
45 }
46
47 // pushWriter adds a writer to the encoder.
48 func (enc *Encoder) pushWriter(w io.Writer) {
49     enc.w = append(enc.w, w)
50 }
51
52 // popWriter pops the innermost writer.
53 func (enc *Encoder) popWriter() {
54     enc.w = enc.w[0 : len(enc.w)-1]
55 }
56
57 func (enc *Encoder) badType(rt reflect.Type) {
58     enc.setError(errors.New("gob: can't encode type " +
59 ))
60
61 func (enc *Encoder) setError(err error) {
62     if enc.err == nil { // remember the first.
63         enc.err = err
64     }
65 }
66
67 // writeMessage sends the data item preceded by a unsigned c
68 func (enc *Encoder) writeMessage(w io.Writer, b *bytes.Buffer) {
69     // Space has been reserved for the length at the head
70     // This is a little dirty: we grab the slice from the head
71     // it by hand.
72     message := b.Bytes()
73     messageLen := len(message) - maxLength
74     // Encode the length.
75     enc.countState.b.Reset()
76     enc.countState.encodeUint(uint64(messageLen))
77     // Copy the length to be a prefix of the message.
78     offset := maxLength - enc.countState.b.Len()
79     copy(message[offset:], enc.countState.b.Bytes())
80     // Write the data.
81     _, err := w.Write(message[offset:])
82     // Drain the buffer and restore the space at the head
83     b.Reset()
84     b.Write(spaceForLength)
85     if err != nil {
86         enc.setError(err)
87     }
88 }
89
90 // sendActualType sends the requested type, without further
91 // it's been sent before.

```

```

92 func (enc *Encoder) sendActualType(w io.Writer, state *encod
93     if _, alreadySent := enc.sent[actual]; alreadySent {
94         return false
95     }
96     typeLock.Lock()
97     info, err := getTypeInfo(ut)
98     typeLock.Unlock()
99     if err != nil {
100         enc.setError(err)
101         return
102     }
103     // Send the pair (-id, type)
104     // Id:
105     state.encodeInt(-int64(info.id))
106     // Type:
107     enc.encode(state.b, reflect.ValueOf(info.wire), wire
108     enc.writeMessage(w, state.b)
109     if enc.err != nil {
110         return
111     }
112
113     // Remember we've sent this type, both what the user
114     enc.sent[ut.base] = info.id
115     if ut.user != ut.base {
116         enc.sent[ut.user] = info.id
117     }
118     // Now send the inner types
119     switch st := actual; st.Kind() {
120     case reflect.Struct:
121         for i := 0; i < st.NumField(); i++ {
122             if isExported(st.Field(i).Name) {
123                 enc.sendType(w, state, st.Fi
124             }
125         }
126     case reflect.Array, reflect.Slice:
127         enc.sendType(w, state, st.Elem())
128     case reflect.Map:
129         enc.sendType(w, state, st.Key())
130         enc.sendType(w, state, st.Elem())
131     }
132     return true
133 }
134
135 // sendType sends the type info to the other side, if necess
136 func (enc *Encoder) sendType(w io.Writer, state *encoderStat
137     ut := userType(origt)
138     if ut.isGobEncoder {
139         // The rules are different: regardless of th
140         // we need to tell the other side that this

```

```

141         return enc.sendActualType(w, state, ut, ut.u
142     }
143
144     // It's a concrete value, so drill down to the base
145     switch rt := ut.base; rt.Kind() {
146     default:
147         // Basic types and interfaces do not need to
148         return
149     case reflect.Slice:
150         // If it's []uint8, don't send; it's conside
151         if rt.Elem().Kind() == reflect.Uint8 {
152             return
153         }
154         // Otherwise we do send.
155         break
156     case reflect.Array:
157         // arrays must be sent so we know their leng
158         break
159     case reflect.Map:
160         // maps must be sent so we know their length
161         break
162     case reflect.Struct:
163         // structs must be sent so we know their fie
164         break
165     case reflect.Chan, reflect.Func:
166         // Probably a bad field in a struct.
167         enc.badType(rt)
168         return
169     }
170
171     return enc.sendActualType(w, state, ut, ut.base)
172 }
173
174 // Encode transmits the data item represented by the empty i
175 // guaranteeing that all necessary type information has been
176 func (enc *Encoder) Encode(e interface{}) error {
177     return enc.EncodeValue(reflect.ValueOf(e))
178 }
179
180 // sendTypeDescriptor makes sure the remote side knows about
181 // It will send a descriptor if this is the first time the t
182 // sent.
183 func (enc *Encoder) sendTypeDescriptor(w io.Writer, state *e
184     // Make sure the type is known to the other side.
185     // First, have we already sent this type?
186     rt := ut.base
187     if ut.isGobEncoder {
188         rt = ut.user
189     }

```

```

190     if _, alreadySent := enc.sent[rt]; !alreadySent {
191         // No, so send it.
192         sent := enc.sendType(w, state, rt)
193         if enc.err != nil {
194             return
195         }
196         // If the type info has still not been trans
197         // a singleton basic type (int, []byte etc.)
198         // need to send the type info but we do need
199         if !sent {
200             typeLock.Lock()
201             info, err := getTypeInfo(ut)
202             typeLock.Unlock()
203             if err != nil {
204                 enc.setError(err)
205                 return
206             }
207             enc.sent[rt] = info.id
208         }
209     }
210 }
211
212 // sendTypeId sends the id, which must have already been def
213 func (enc *Encoder) sendTypeId(state *encoderState, ut *user
214     // Identify the type of this top-level value.
215     state.encodeInt(int64(enc.sent[ut.base]))
216 }
217
218 // EncodeValue transmits the data item represented by the re
219 // guaranteeing that all necessary type information has been
220 func (enc *Encoder) EncodeValue(value reflect.Value) error {
221     // Make sure we're single-threaded through here, so
222     // goroutines can share an encoder.
223     enc.mutex.Lock()
224     defer enc.mutex.Unlock()
225
226     // Remove any nested writers remaining due to previo
227     enc.w = enc.w[0:1]
228
229     ut, err := validUserType(value.Type())
230     if err != nil {
231         return err
232     }
233
234     enc.err = nil
235     enc.byteBuf.Reset()
236     enc.byteBuf.Write(spaceForLength)
237     state := enc.newEncoderState(&enc.byteBuf)
238
239     enc.sendTypeDescriptor(enc.writer(), state, ut)

```

```
240     enc.sendTypeId(state, ut)
241     if enc.err != nil {
242         return enc.err
243     }
244
245     // Encode the object.
246     enc.encode(state.b, value, ut)
247     if enc.err == nil {
248         enc.writeMessage(enc.writer(), state.b)
249     }
250
251     enc.freeEncoderState(state)
252     return enc.err
253 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/gob/error.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package gob
6
7 import "fmt"
8
9 // Errors in decoding and encoding are handled using panic a
10 // Panics caused by user error (that is, everything except r
11 // such as "index out of bounds" errors) do not leave the fi
12 // them, but are instead turned into plain error returns. E
13 // decoding functions and methods that do not return an erro
14 // panic to report an error or are guaranteed error-free.
15
16 // A gobError is used to distinguish errors (panics) generat
17 type gobError struct {
18     err error
19 }
20
21 // errorf is like error_ but takes Printf-style arguments to
22 // It always prefixes the message with "gob: ".
23 func errorf(format string, args ...interface{}) {
24     error_(fmt.Errorf("gob: "+format, args...))
25 }
26
27 // error wraps the argument error and uses it as the argumen
28 func error_(err error) {
29     panic(gobError{err})
30 }
31
32 // catchError is meant to be used as a deferred function to
33 // plain error. It overwrites the error return of the funct
34 func catchError(err *error) {
35     if e := recover(); e != nil {
36         ge, ok := e.(gobError)
37         if !ok {
38             panic(e)
39         }
40         *err = ge.err
41     }
42 }
```

```
42         return  
43     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/gob/type.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package gob
6
7 import (
8     "errors"
9     "fmt"
10    "os"
11    "reflect"
12    "sync"
13    "unicode"
14    "unicode/utf8"
15 )
16
17 // userTypeInfo stores the information associated with a type
18 // to the package. It's computed once and stored in a map k
19 // type.
20 type userTypeInfo struct {
21     user      reflect.Type // the type the user handles
22     base      reflect.Type // the base type after all
23     indir     int          // number of indirections
24     isGobEncoder bool        // does the type implement
25     isGobDecoder bool        // does the type implement
26     encIndir  int8        // number of indirections
27     decIndir  int8        // number of indirections
28 }
29
30 var (
31     // Protected by an RWMutex because we read it a lot
32     // it only when we see a new type, typically when co
33     userTypeLock sync.RWMutex
34     userTypeCache = make(map[reflect.Type]*userTypeInfo)
35 )
36
37 // validType returns, and saves, the information associated
38 // If the user type is not valid, err will be non-nil. To b
39 // is not set up.
40 func validUserType(rt reflect.Type) (ut *userTypeInfo, err error) {
41     userTypeLock.RLock()
```

```

42         ut = userTypeCache[rt]
43         userTypeLock.RUnlock()
44         if ut != nil {
45             return
46         }
47         // Now set the value under the write lock.
48         userTypeLock.Lock()
49         defer userTypeLock.Unlock()
50         if ut = userTypeCache[rt]; ut != nil {
51             // Lost the race; not a problem.
52             return
53         }
54         ut = new(userTypeInfo)
55         ut.base = rt
56         ut.user = rt
57         // A type that is just a cycle of pointers (such as
58         // be represented in gobs, which need some concrete
59         // cycle detection algorithm from Knuth, Vol 2, Sect
60         // pp 539-540. As we step through indirections, run
61         // half speed. If they meet up, there's a cycle.
62         slowpoke := ut.base // walks half as fast as ut.base
63         for {
64             pt := ut.base
65             if pt.Kind() != reflect.Ptr {
66                 break
67             }
68             ut.base = pt.Elem()
69             if ut.base == slowpoke { // ut.base lapped s
70                 // recursive pointer type.
71                 return nil, errors.New("can't repres
72             }
73             if ut.indir%2 == 0 {
74                 slowpoke = slowpoke.Elem()
75             }
76             ut.indir++
77         }
78         ut.isGobEncoder, ut.encIndir = implementsInterface(u
79         ut.isGobDecoder, ut.decIndir = implementsInterface(u
80         userTypeCache[rt] = ut
81         return
82     }
83
84     var (
85         gobEncoderInterfaceType = reflect.TypeOf((*GobEncode
86         gobDecoderInterfaceType = reflect.TypeOf((*GobDecode
87     )
88
89     // implementsInterface reports whether the type implements t
90     // gobEncoder/gobDecoder interface.
91     // It also returns the number of indirections required to ge

```

```

92 // implementation.
93 func implementsInterface(typ, gobEncDecType reflect.Type) (s
94     if typ == nil {
95         return
96     }
97     rt := typ
98     // The type might be a pointer and we need to keep
99     // dereferencing to the base type until we find an i
100    for {
101        if rt.Implements(gobEncDecType) {
102            return true, indir
103        }
104        if p := rt; p.Kind() == reflect.Ptr {
105            indir++
106            if indir > 100 { // insane number of
107                return false, 0
108            }
109            rt = p.Elem()
110            continue
111        }
112        break
113    }
114    // No luck yet, but if this is a base type (non-poin
115    if typ.Kind() != reflect.Ptr {
116        // Not a pointer, but does the pointer work?
117        if reflect.PtrTo(typ).Implements(gobEncDecTy
118            return true, -1
119        }
120    }
121    return false, 0
122 }
123
124 // userType returns, and saves, the information associated w
125 // If the user type is not valid, it calls error.
126 func userType(rt reflect.Type) *userTypeInfo {
127     ut, err := validUserType(rt)
128     if err != nil {
129         error_(err)
130     }
131     return ut
132 }
133
134 // A typeId represents a gob Type as an integer that can be
135 // Internally, typeIds are used as keys to a map to recover
136 type typeId int32
137
138 var nextId typeId // incremented for each new type we
139 var typeLock sync.Mutex // set while building a type
140 const firstUserId = 64 // lowest id number granted to user

```

```

141
142 type gobType interface {
143     id() typeId
144     setId(id typeId)
145     name() string
146     string() string // not public; only for debugging
147     safeString(seen map[typeId]bool) string
148 }
149
150 var types = make(map[reflect.Type]gobType)
151 var idToType = make(map[typeId]gobType)
152 var builtinIdToType map[typeId]gobType // set in init() afte
153
154 func setId(typ gobType) {
155     // When building recursive types, someone may get th
156     if typ.id() != 0 {
157         return
158     }
159     nextId++
160     typ.setId(nextId)
161     idToType[nextId] = typ
162 }
163
164 func (t typeId) gobType() gobType {
165     if t == 0 {
166         return nil
167     }
168     return idToType[t]
169 }
170
171 // string returns the string representation of the type asso
172 func (t typeId) string() string {
173     if t.gobType() == nil {
174         return "<nil>"
175     }
176     return t.gobType().string()
177 }
178
179 // Name returns the name of the type associated with the typ
180 func (t typeId) name() string {
181     if t.gobType() == nil {
182         return "<nil>"
183     }
184     return t.gobType().name()
185 }
186
187 // CommonType holds elements of all types.
188 // It is a historical artifact, kept for binary compatibilit
189 // only for the benefit of the package's encoding of type de

```

```

190 // not intended for direct use by clients.
191 type CommonType struct {
192     Name string
193     Id    typeId
194 }
195
196 func (t *CommonType) id() typeId { return t.Id }
197
198 func (t *CommonType) setId(id typeId) { t.Id = id }
199
200 func (t *CommonType) string() string { return t.Name }
201
202 func (t *CommonType) safeString(seen map[typeId]bool) string
203     return t.Name
204 }
205
206 func (t *CommonType) name() string { return t.Name }
207
208 // Create and check predefined types
209 // The string for tBytes is "bytes" not "[]byte" to signify
210
211 var (
212     // Primordial types, needed during initialization.
213     // Always passed as pointers so the interface{} type
214     // goes through without losing its interfaceness.
215     tBool    = bootstrapType("bool", (*bool)(nil), 1)
216     tInt     = bootstrapType("int", (*int)(nil), 2)
217     tUint    = bootstrapType("uint", (*uint)(nil), 3)
218     tFloat   = bootstrapType("float", (*float64)(nil),
219     tBytes   = bootstrapType("bytes", (*[]byte)(nil),
220     tString  = bootstrapType("string", (*string)(nil),
221     tComplex = bootstrapType("complex", (*complex128)(
222     tInterface = bootstrapType("interface", (*interface{
223     // Reserve some Ids for compatible expansion
224     tReserved7 = bootstrapType("_reserved1", (*struct{ r
225     tReserved6 = bootstrapType("_reserved1", (*struct{ r
226     tReserved5 = bootstrapType("_reserved1", (*struct{ r
227     tReserved4 = bootstrapType("_reserved1", (*struct{ r
228     tReserved3 = bootstrapType("_reserved1", (*struct{ r
229     tReserved2 = bootstrapType("_reserved1", (*struct{ r
230     tReserved1 = bootstrapType("_reserved1", (*struct{ r
231 )
232
233 // Predefined because it's needed by the Decoder
234 var tWireType = mustGetTypeInfo(reflect.TypeOf(wireType{})).
235 var wireTypeUserInfo *userInfo // userInfo of (*wire
236
237 func init() {
238     // Some magic numbers to make sure there are no surp
239     checkId(16, tWireType)

```

```

240     checkId(17, mustGetTypeInfo(reflect.TypeOf(arrayType
241     checkId(18, mustGetTypeInfo(reflect.TypeOf(CommonTyp
242     checkId(19, mustGetTypeInfo(reflect.TypeOf(sliceType
243     checkId(20, mustGetTypeInfo(reflect.TypeOf(structTyp
244     checkId(21, mustGetTypeInfo(reflect.TypeOf(fieldType
245     checkId(23, mustGetTypeInfo(reflect.TypeOf(mapType{}
246
247     builtinIdToType = make(map[typeId]gobType)
248     for k, v := range idToType {
249         builtinIdToType[k] = v
250     }
251
252     // Move the id space upwards to allow for growth in
253     // without breaking existing files.
254     if nextId > firstUserId {
255         panic(fmt.Sprintf("nextId too large:", next
256     }
257     nextId = firstUserId
258     registerBasics()
259     wireTypeUserInfo = userType(reflect.TypeOf((*wireTyp
260 }
261
262 // Array type
263 type arrayType struct {
264     CommonType
265     Elem typeId
266     Len int
267 }
268
269 func newArrayType(name string) *arrayType {
270     a := &arrayType{CommonType{Name: name}, 0, 0}
271     return a
272 }
273
274 func (a *arrayType) init(elem gobType, len int) {
275     // Set our type id before evaluating the element's,
276     setId(a)
277     a.Elem = elem.id()
278     a.Len = len
279 }
280
281 func (a *arrayType) safeString(seen map[typeId]bool) string
282     if seen[a.Id] {
283         return a.Name
284     }
285     seen[a.Id] = true
286     return fmt.Sprintf("[%d]s", a.Len, a.Elem.gobType())
287 }
288

```

```

289 func (a *arrayType) string() string { return a.safeString(ma
290
291 // GobEncoder type (something that implements the GobEncoder
292 type gobEncoderType struct {
293     CommonType
294 }
295
296 func newGobEncoderType(name string) *gobEncoderType {
297     g := &gobEncoderType{CommonType{Name: name}}
298     setTypeId(g)
299     return g
300 }
301
302 func (g *gobEncoderType) safeString(seen map[typeId]bool) st
303     return g.Name
304 }
305
306 func (g *gobEncoderType) string() string { return g.Name }
307
308 // Map type
309 type mapType struct {
310     CommonType
311     Key typeId
312     Elem typeId
313 }
314
315 func newMapType(name string) *mapType {
316     m := &mapType{CommonType{Name: name}, 0, 0}
317     return m
318 }
319
320 func (m *mapType) init(key, elem gobType) {
321     // Set our type id before evaluating the element's,
322     setTypeId(m)
323     m.Key = key.id()
324     m.Elem = elem.id()
325 }
326
327 func (m *mapType) safeString(seen map[typeId]bool) string {
328     if seen[m.Id] {
329         return m.Name
330     }
331     seen[m.Id] = true
332     key := m.Key.gobType().safeString(seen)
333     elem := m.Elem.gobType().safeString(seen)
334     return fmt.Sprintf("map[%s]%s", key, elem)
335 }
336
337 func (m *mapType) string() string { return m.safeString(make

```

```

338
339 // Slice type
340 type sliceType struct {
341     CommonType
342     Elem typeId
343 }
344
345 func newSliceType(name string) *sliceType {
346     s := &sliceType{CommonType{Name: name}, 0}
347     return s
348 }
349
350 func (s *sliceType) init(elem gobType) {
351     // Set our type id before evaluating the element's,
352     setTypeId(s)
353     // See the comments about ids in newTypeObject. Only
354     // structs have mutual recursion.
355     if elem.id() == 0 {
356         setTypeId(elem)
357     }
358     s.Elem = elem.id()
359 }
360
361 func (s *sliceType) safeString(seen map[typeId]bool) string
362     if seen[s.Id] {
363         return s.Name
364     }
365     seen[s.Id] = true
366     return fmt.Sprintf("[%s", s.Elem.gobType().safeStri
367 }
368
369 func (s *sliceType) string() string { return s.safeString(ma
370
371 // Struct type
372 type fieldType struct {
373     Name string
374     Id    typeId
375 }
376
377 type structType struct {
378     CommonType
379     Field []*fieldType
380 }
381
382 func (s *structType) safeString(seen map[typeId]bool) string
383     if s == nil {
384         return "<nil>"
385     }
386     if _, ok := seen[s.Id]; ok {
387         return s.Name

```

```

388     }
389     seen[s.Id] = true
390     str := s.Name + " = struct { "
391     for _, f := range s.Field {
392         str += fmt.Sprintf("%s %s; ", f.Name, f.Id.g
393     }
394     str += "}"
395     return str
396 }
397
398 func (s *structType) string() string { return s.safeString(m
399
400 func newStructType(name string) *structType {
401     s := &structType{CommonType{Name: name}, nil}
402     // For historical reasons we set the id here rather
403     // See the comment in newTypeObject for details.
404     setTypeId(s)
405     return s
406 }
407
408 // newTypeObject allocates a gobType for the reflection type
409 // Unless ut represents a GobEncoder, rt should be the base
410 // of ut.
411 // This is only called from the encoding side. The decoding
412 // works through typeIdS and userTypeInfos alone.
413 func newTypeObject(name string, ut *userTypeInfo, rt reflect
414     // Does this type implement GobEncoder?
415     if ut.isGobEncoder {
416         return newGobEncoderType(name), nil
417     }
418     var err error
419     var type0, type1 gobType
420     defer func() {
421         if err != nil {
422             delete(types, rt)
423         }
424     }()
425     // Install the top-level type before the subtypes (e
426     // fields) so recursive types can be constructed saf
427     switch t := rt; t.Kind() {
428     // All basic types are easy: they are predefined.
429     case reflect.Bool:
430         return t.Bool.gobType(), nil
431
432     case reflect.Int, reflect.Int8, reflect.Int16, refle
433         return t.Int.gobType(), nil
434
435     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
436         return t.Uint.gobType(), nil

```

```

437
438     case reflect.Float32, reflect.Float64:
439         return tFloat.gobType(), nil
440
441     case reflect.Complex64, reflect.Complex128:
442         return tComplex.gobType(), nil
443
444     case reflect.String:
445         return tString.gobType(), nil
446
447     case reflect.Interface:
448         return tInterface.gobType(), nil
449
450     case reflect.Array:
451         at := newArrayType(name)
452         types[rt] = at
453         type0, err = getBaseType("", t.Elem())
454         if err != nil {
455             return nil, err
456         }
457         // Historical aside:
458         // For arrays, maps, and slices, we set the
459         // are constructed. This is to retain the or
460         // a fix made to handle recursive types, whi
461         // which types are built. Delaying the sett
462         // type ids while allowing recursive types t
463         // done below, were already handling recursi
464         // assign the top-level id before those of t
465         at.init(type0, t.Len())
466         return at, nil
467
468     case reflect.Map:
469         mt := newMapType(name)
470         types[rt] = mt
471         type0, err = getBaseType("", t.Key())
472         if err != nil {
473             return nil, err
474         }
475         type1, err = getBaseType("", t.Elem())
476         if err != nil {
477             return nil, err
478         }
479         mt.init(type0, type1)
480         return mt, nil
481
482     case reflect.Slice:
483         // []byte == []uint8 is a special case
484         if t.Elem().Kind() == reflect.Uint8 {
485             return tBytes.gobType(), nil

```

```

486     }
487     st := newSliceType(name)
488     types[rt] = st
489     type0, err = getBaseType(t.Elem().Name(), t.
490     if err != nil {
491         return nil, err
492     }
493     st.init(type0)
494     return st, nil
495
496     case reflect.Struct:
497         st := newStructType(name)
498         types[rt] = st
499         idToType[st.id()] = st
500         for i := 0; i < t.NumField(); i++ {
501             f := t.Field(i)
502             if !isExported(f.Name) {
503                 continue
504             }
505             typ := userType(f.Type).base
506             tname := typ.Name()
507             if tname == "" {
508                 t := userType(f.Type).base
509                 tname = t.String()
510             }
511             gt, err := getBaseType(tname, f.Type)
512             if err != nil {
513                 return nil, err
514             }
515             // Some mutually recursive types can
516             // still defining the element. Fix t
517             // We could do this more neatly by s
518             // building every type, but that wou
519             if gt.id() == 0 {
520                 setTypeId(gt)
521             }
522             st.Field = append(st.Field, &fieldTy
523         }
524         return st, nil
525
526     default:
527         return nil, errors.New("gob NewTypeObject ca
528     }
529     return nil, nil
530 }
531
532 // isExported reports whether this is an exported - upper ca
533 func isExported(name string) bool {
534     rune, _ := utf8.DecodeRuneInString(name)
535     return unicode.IsUpper(rune)

```

```

536 }
537
538 // getBaseType returns the Gob type describing the given ref
539 // typeLock must be held.
540 func getBaseType(name string, rt reflect.Type) (gobType, err
541     ut := userType(rt)
542     return getType(name, ut, ut.base)
543 }
544
545 // getType returns the Gob type describing the given reflect
546 // Should be called only when handling GobEncoders/Decoders,
547 // which may be pointers. All other types are handled throu
548 // base type, never a pointer.
549 // typeLock must be held.
550 func getType(name string, ut *userTypeInfo, rt reflect.Type)
551     typ, present := types[rt]
552     if present {
553         return typ, nil
554     }
555     typ, err := newTypeObject(name, ut, rt)
556     if err == nil {
557         types[rt] = typ
558     }
559     return typ, err
560 }
561
562 func checkId(want, got typeId) {
563     if want != got {
564         fmt.Fprintf(os.Stderr, "checkId: %d should b
565         panic("bootstrap type wrong id: " + got.name
566     }
567 }
568
569 // used for building the basic types; called only from init(
570 // interface always refers to a pointer.
571 func bootstrapType(name string, e interface{}, expect typeId
572     rt := reflect.TypeOf(e).Elem()
573     _, present := types[rt]
574     if present {
575         panic("bootstrap type already present: " + n
576     }
577     typ := &CommonType{Name: name}
578     types[rt] = typ
579     setTypeId(typ)
580     checkId(expect, nextId)
581     userType(rt) // might as well cache it now
582     return nextId
583 }
584

```

```

585 // Representation of the information we send and receive abo
586 // Each value we send is preceded by its type definition: an
587 // However, the very first time we send the value, we first
588 // (-id, wireType).
589 // For bootstrapping purposes, we assume that the recipient
590 // to decode a wireType; it is exactly the wireType struct h
591 // using the gob rules for sending a structure, except that
592 // ids for wireType and structType etc. are known. The rele
593 // are built in encode.go's init() function.
594 // To maintain binary compatibility, if you extend this type
595 // the new fields last.
596 type wireType struct {
597     ArrayT      *arrayType
598     SliceT      *sliceType
599     StructT     *structType
600     MapT        *mapType
601     GobEncoderT *gobEncoderType
602 }
603
604 func (w *wireType) string() string {
605     const unknown = "unknown type"
606     if w == nil {
607         return unknown
608     }
609     switch {
610     case w.ArrayT != nil:
611         return w.ArrayT.Name
612     case w.SliceT != nil:
613         return w.SliceT.Name
614     case w.StructT != nil:
615         return w.StructT.Name
616     case w.MapT != nil:
617         return w.MapT.Name
618     case w.GobEncoderT != nil:
619         return w.GobEncoderT.Name
620     }
621     return unknown
622 }
623
624 type typeInfo struct {
625     id      typeId
626     encoder *encEngine
627     wire    *wireType
628 }
629
630 var typeInfoMap = make(map[reflect.Type]*typeInfo) // protec
631
632 // typeLock must be held.
633 func getTypeInfo(ut *userTypeInfo) (*typeInfo, error) {

```

```

634     rt := ut.base
635     if ut.isGobEncoder {
636         // We want the user type, not the base type.
637         rt = ut.user
638     }
639     info, ok := typeInfoMap[rt]
640     if ok {
641         return info, nil
642     }
643     info = new(typeInfo)
644     gt, err := getBaseType(rt.Name(), rt)
645     if err != nil {
646         return nil, err
647     }
648     info.id = gt.id()
649
650     if ut.isGobEncoder {
651         userType, err := getType(rt.Name(), ut, rt)
652         if err != nil {
653             return nil, err
654         }
655         info.wire = &wireType{GobEncoderT: userType.
656         typeInfoMap[ut.user] = info
657         return info, nil
658     }
659
660     t := info.id.gobType()
661     switch typ := rt; typ.Kind() {
662     case reflect.Array:
663         info.wire = &wireType{ArrayT: t.(*arrayType)}
664     case reflect.Map:
665         info.wire = &wireType{MapT: t.(*mapType)}
666     case reflect.Slice:
667         // []byte == []uint8 is a special case handl
668         if typ.Elem().Kind() != reflect.Uint8 {
669             info.wire = &wireType{SliceT: t.(*sl
670         }
671     case reflect.Struct:
672         info.wire = &wireType{StructT: t.(*structTyp
673     }
674     typeInfoMap[rt] = info
675     return info, nil
676 }
677
678 // Called only when a panic is acceptable and unexpected.
679 func mustGetTypeInfo(rt reflect.Type) *typeInfo {
680     t, err := getTypeInfo(userType(rt))
681     if err != nil {
682         panic("getTypeInfo: " + err.Error())
683     }

```

```

684         return t
685     }
686
687     // GobEncoder is the interface describing data that provides
688     // representation for encoding values for transmission to a
689     // A type that implements GobEncoder and GobDecoder has comp
690     // control over the representation of its data and may there
691     // contain things such as private fields, channels, and func
692     // which are not usually transmissible in gob streams.
693     //
694     // Note: Since gobs can be stored permanently, It is good de
695     // to guarantee the encoding used by a GobEncoder is stable
696     // software evolves. For instance, it might make sense for
697     // to include a version number in the encoding.
698     type GobEncoder interface {
699         // GobEncode returns a byte slice representing the e
700         // receiver for transmission to a GobDecoder, usuall
701         // concrete type.
702         GobEncode() ([]byte, error)
703     }
704
705     // GobDecoder is the interface describing data that provides
706     // routine for decoding transmitted values sent by a GobEnco
707     type GobDecoder interface {
708         // GobDecode overwrites the receiver, which must be
709         // with the value represented by the byte slice, whi
710         // by GobEncode, usually for the same concrete type.
711         GobDecode([]byte) error
712     }
713
714     var (
715         nameToConcreteType = make(map[string]reflect.Type)
716         concreteTypeToName = make(map[reflect.Type]string)
717     )
718
719     // RegisterName is like Register but uses the provided name
720     // type's default.
721     func RegisterName(name string, value interface{}) {
722         if name == "" {
723             // reserved for nil
724             panic("attempt to register empty name")
725         }
726         ut := userType(reflect.TypeOf(value))
727         // Check for incompatible duplicates. The name must
728         // same user type, and vice versa.
729         if t, ok := nameToConcreteType[name]; ok && t != ut.
730             panic(fmt.Sprintf("gob: registering duplicat
731         }
732         if n, ok := concreteTypeToName[ut.base]; ok && n !=

```

```

733         panic(fmt.Sprintf("gob: registering duplicat
734     }
735     // Store the name and type provided by the user....
736     nameToConcreteType[name] = reflect.TypeOf(value)
737     // but the flattened type in the type table, since t
738     concreteTypeToName[ut.base] = name
739 }
740
741 // Register records a type, identified by a value for that t
742 // internal type name. That name will identify the concrete
743 // sent or received as an interface variable. Only types th
744 // transferred as implementations of interface values need t
745 // Expecting to be used only during initialization, it panic
746 // between types and names is not a bijection.
747 func Register(value interface{}) {
748     // Default to printed representation for unnamed typ
749     rt := reflect.TypeOf(value)
750     name := rt.String()
751
752     // But for named types (or pointers to them), qualif
753     // Dereference one pointer looking for a named type.
754     star := ""
755     if rt.Name() == "" {
756         if pt := rt; pt.Kind() == reflect.Ptr {
757             star = "*"
758             rt = pt
759         }
760     }
761     if rt.Name() != "" {
762         if rt.PkgPath() == "" {
763             name = star + rt.Name()
764         } else {
765             name = star + rt.PkgPath() + "." + r
766         }
767     }
768
769     RegisterName(name, value)
770 }
771
772 func registerBasics() {
773     Register(int(0))
774     Register(int8(0))
775     Register(int16(0))
776     Register(int32(0))
777     Register(int64(0))
778     Register(uint(0))
779     Register(uint8(0))
780     Register(uint16(0))
781     Register(uint32(0))

```

```
782     Register(uint64(0))
783     Register(float32(0))
784     Register(float64(0))
785     Register(complex64(0i))
786     Register(complex128(0i))
787     Register(uintptr(0))
788     Register(false)
789     Register("")
790     Register([]byte(nil))
791     Register([]int(nil))
792     Register([]int8(nil))
793     Register([]int16(nil))
794     Register([]int32(nil))
795     Register([]int64(nil))
796     Register([]uint(nil))
797     Register([]uint8(nil))
798     Register([]uint16(nil))
799     Register([]uint32(nil))
800     Register([]uint64(nil))
801     Register([]float32(nil))
802     Register([]float64(nil))
803     Register([]complex64(nil))
804     Register([]complex128(nil))
805     Register([]uintptr(nil))
806     Register([]bool(nil))
807     Register([]string(nil))
808 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/hex/hex.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package hex implements hexadecimal encoding and decoding.
6 package hex
7
8 import (
9     "bytes"
10    "errors"
11    "fmt"
12    "io"
13 )
14
15 const hextable = "0123456789abcdef"
16
17 // EncodedLen returns the length of an encoding of n source
18 func EncodedLen(n int) int { return n * 2 }
19
20 // Encode encodes src into EncodedLen(len(src))
21 // bytes of dst. As a convenience, it returns the number
22 // of bytes written to dst, but this value is always Encoded
23 // Encode implements hexadecimal encoding.
24 func Encode(dst, src []byte) int {
25     for i, v := range src {
26         dst[i*2] = hextable[v>>4]
27         dst[i*2+1] = hextable[v&0x0f]
28     }
29
30     return len(src) * 2
31 }
32
33 // ErrLength results from decoding an odd length slice.
34 var ErrLength = errors.New("encoding/hex: odd length hex str")
35
36 // InvalidByteError values describe errors resulting from an
37 type InvalidByteError byte
38
39 func (e InvalidByteError) Error() string {
40     return fmt.Sprintf("encoding/hex: invalid byte: %#U")
41 }
```

```

42
43 func DecodedLen(x int) int { return x / 2 }
44
45 // Decode decodes src into DecodedLen(len(src)) bytes, retur
46 // number of bytes written to dst.
47 //
48 // If Decode encounters invalid input, it returns an error d
49 func Decode(dst, src []byte) (int, error) {
50     if len(src)%2 == 1 {
51         return 0, ErrLength
52     }
53
54     for i := 0; i < len(src)/2; i++ {
55         a, ok := fromHexChar(src[i*2])
56         if !ok {
57             return 0, InvalidByteError(src[i*2])
58         }
59         b, ok := fromHexChar(src[i*2+1])
60         if !ok {
61             return 0, InvalidByteError(src[i*2+1
62         }
63         dst[i] = (a << 4) | b
64     }
65
66     return len(src) / 2, nil
67 }
68
69 // fromHexChar converts a hex character into its value and a
70 func fromHexChar(c byte) (byte, bool) {
71     switch {
72     case '0' <= c && c <= '9':
73         return c - '0', true
74     case 'a' <= c && c <= 'f':
75         return c - 'a' + 10, true
76     case 'A' <= c && c <= 'F':
77         return c - 'A' + 10, true
78     }
79
80     return 0, false
81 }
82
83 // EncodeToString returns the hexadecimal encoding of src.
84 func EncodeToString(src []byte) string {
85     dst := make([]byte, EncodedLen(len(src)))
86     Encode(dst, src)
87     return string(dst)
88 }
89
90 // DecodeString returns the bytes represented by the hexadec
91 func DecodeString(s string) ([]byte, error) {

```

```

92         src := []byte(s)
93         dst := make([]byte, DecodedLen(len(src)))
94         _, err := Decode(dst, src)
95         if err != nil {
96             return nil, err
97         }
98         return dst, nil
99     }
100
101 // Dump returns a string that contains a hex dump of the giv
102 // of the hex dump matches the output of `hexdump -C` on the
103 func Dump(data []byte) string {
104     var buf bytes.Buffer
105     dumper := Dumper(&buf)
106     dumper.Write(data)
107     dumper.Close()
108     return string(buf.Bytes())
109 }
110
111 // Dumper returns a WriteCloser that writes a hex dump of al
112 // w. The format of the dump matches the output of `hexdump
113 // line.
114 func Dumper(w io.Writer) io.WriteCloser {
115     return &dumper{w: w}
116 }
117
118 type dumper struct {
119     w          io.Writer
120     rightChars [18]byte
121     buf        [14]byte
122     used       int // number of bytes in the current li
123     n          uint // number of bytes, total
124 }
125
126 func toChar(b byte) byte {
127     if b < 32 || b > 126 {
128         return '.'
129     }
130     return b
131 }
132
133 func (h *dumper) Write(data []byte) (n int, err error) {
134     // Output lines look like:
135     // 00000010 2e 2f 30 31 32 33 34 35 36 37 38 39 3a
136     // ^ offset                                     ^ extra space
137     for i := range data {
138         if h.used == 0 {
139             // At the beginning of a line we pri
140             // offset in hex.

```

```

141         h.buf[0] = byte(h.n >> 24)
142         h.buf[1] = byte(h.n >> 16)
143         h.buf[2] = byte(h.n >> 8)
144         h.buf[3] = byte(h.n)
145         Encode(h.buf[4:], h.buf[:4])
146         h.buf[12] = ' '
147         h.buf[13] = ' '
148         _, err = h.w.Write(h.buf[4:])
149     }
150     Encode(h.buf[:], data[i:i+1])
151     h.buf[2] = ' '
152     l := 3
153     if h.used == 7 {
154         // There's an additional space after
155         h.buf[3] = ' '
156         l = 4
157     } else if h.used == 15 {
158         // At the end of the line there's an
159         // the bar for the right column.
160         h.buf[3] = ' '
161         h.buf[4] = '|'
162         l = 5
163     }
164     _, err = h.w.Write(h.buf[:l])
165     if err != nil {
166         return
167     }
168     n++
169     h.rightChars[h.used] = toChar(data[i])
170     h.used++
171     h.n++
172     if h.used == 16 {
173         h.rightChars[16] = '|'
174         h.rightChars[17] = '\n'
175         _, err = h.w.Write(h.rightChars[:])
176         if err != nil {
177             return
178         }
179         h.used = 0
180     }
181 }
182 return
183 }
184
185 func (h *dumper) Close() (err error) {
186     // See the comments in Write() for the details of th
187     if h.used == 0 {
188         return
189     }

```

```

190         h.buf[0] = ' '
191         h.buf[1] = ' '
192         h.buf[2] = ' '
193         h.buf[3] = ' '
194         h.buf[4] = '|'
195         nBytes := h.used
196         for h.used < 16 {
197             l := 3
198             if h.used == 7 {
199                 l = 4
200             } else if h.used == 15 {
201                 l = 5
202             }
203             _, err = h.w.Write(h.buf[:l])
204             if err != nil {
205                 return
206             }
207             h.used++
208         }
209         h.rightChars[nBytes] = '|'
210         h.rightChars[nBytes+1] = '\n'
211         _, err = h.w.Write(h.rightChars[:nBytes+2])
212         return
213     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/json/decode.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Represents JSON data structure using native Go types: boo
6 // strings, arrays, and maps.
7
8 package json
9
10 import (
11     "encoding/base64"
12     "errors"
13     "fmt"
14     "reflect"
15     "runtime"
16     "strconv"
17     "strings"
18     "unicode"
19     "unicode/utf16"
20     "unicode/utf8"
21 )
22
23 // Unmarshal parses the JSON-encoded data and stores the res
24 // in the value pointed to by v.
25 //
26 // Unmarshal uses the inverse of the encodings that
27 // Marshal uses, allocating maps, slices, and pointers as ne
28 // with the following additional rules:
29 //
30 // To unmarshal JSON into a pointer, Unmarshal first handles
31 // the JSON being the JSON literal null. In that case, Unma
32 // the pointer to nil. Otherwise, Unmarshal unmarshals the
33 // the value pointed at by the pointer. If the pointer is n
34 // allocates a new value for it to point to.
35 //
36 // To unmarshal JSON into an interface value, Unmarshal unma
37 // the JSON into the concrete value contained in the interfa
38 // If the interface value is nil, that is, has no concrete v
39 // Unmarshal stores one of these in the interface value:
40 //
41 //     bool, for JSON booleans
```

```

42 //      float64, for JSON numbers
43 //      string, for JSON strings
44 //      []interface{}, for JSON arrays
45 //      map[string]interface{}, for JSON objects
46 //      nil for JSON null
47 //
48 // If a JSON value is not appropriate for a given target typ
49 // or if a JSON number overflows the target type, Unmarshal
50 // skips that field and completes the unmarshalling as best
51 // If no more serious errors are encountered, Unmarshal retu
52 // an UnmarshalTypeError describing the earliest such error.
53 //
54 func Unmarshal(data []byte, v interface{}) error {
55     d := new(decodeState).init(data)
56
57     // Quick check for well-formedness.
58     // Avoids filling out half a data structure
59     // before discovering a JSON syntax error.
60     err := checkValid(data, &d.scan)
61     if err != nil {
62         return err
63     }
64
65     return d.unmarshal(v)
66 }
67
68 // Unmarshaler is the interface implemented by objects
69 // that can unmarshal a JSON description of themselves.
70 // The input can be assumed to be a valid JSON object
71 // encoding. UnmarshalJSON must copy the JSON data
72 // if it wishes to retain the data after returning.
73 type Unmarshaler interface {
74     UnmarshalJSON([]byte) error
75 }
76
77 // An UnmarshalTypeError describes a JSON value that was
78 // not appropriate for a value of a specific Go type.
79 type UnmarshalTypeError struct {
80     Value string // description of JSON value - "b
81     Type  reflect.Type // type of Go value it could not
82 }
83
84 func (e *UnmarshalTypeError) Error() string {
85     return "json: cannot unmarshal " + e.Value + " into
86 }
87
88 // An UnmarshalFieldError describes a JSON object key that
89 // led to an unexported (and therefore unwritable) struct fi
90 type UnmarshalFieldError struct {
91     Key  string

```

```

92         Type reflect.Type
93         Field reflect.StructField
94     }
95
96     func (e *UnmarshalFieldError) Error() string {
97         return "json: cannot unmarshal object key " + strconv
98     }
99
100    // An InvalidUnmarshalError describes an invalid argument pa
101    // (The argument to Unmarshal must be a non-nil pointer.)
102    type InvalidUnmarshalError struct {
103        Type reflect.Type
104    }
105
106    func (e *InvalidUnmarshalError) Error() string {
107        if e.Type == nil {
108            return "json: Unmarshal(nil)"
109        }
110
111        if e.Type.Kind() != reflect.Ptr {
112            return "json: Unmarshal(non-pointer " + e.Type
113        }
114        return "json: Unmarshal(nil " + e.Type.String() + ")"
115    }
116
117    func (d *decodeState) unmarshal(v interface{}) (err error) {
118        defer func() {
119            if r := recover(); r != nil {
120                if _, ok := r.(runtime.Error); ok {
121                    panic(r)
122                }
123                err = r.(error)
124            }
125        }()
126
127        rv := reflect.ValueOf(v)
128        pv := rv
129        if pv.Kind() != reflect.Ptr || pv.IsNil() {
130            return &InvalidUnmarshalError{reflect.TypeOf
131        }
132
133        d.scan.reset()
134        // We decode rv not pv.Elem because the Unmarshaler
135        // test must be applied at the top level of the value
136        d.value(rv)
137        return d.savedError
138    }
139
140    // decodeState represents the state while decoding a JSON va

```

```

141 type decodeState struct {
142     data      []byte
143     off       int // read offset in data
144     scan      scanner
145     nextscan  scanner // for calls to nextValue
146     savedError error
147     tempstr   string // scratch space to avoid some all
148 }
149
150 // errPhase is used for errors that should not happen unless
151 // there is a bug in the JSON decoder or something is editin
152 // the data slice while the decoder executes.
153 var errPhase = errors.New("JSON decoder out of sync - data c
154
155 func (d *decodeState) init(data []byte) *decodeState {
156     d.data = data
157     d.off = 0
158     d.savedError = nil
159     return d
160 }
161
162 // error aborts the decoding by panicking with err.
163 func (d *decodeState) error(err error) {
164     panic(err)
165 }
166
167 // saveError saves the first err it is called with,
168 // for reporting at the end of the unmarshal.
169 func (d *decodeState) saveError(err error) {
170     if d.savedError == nil {
171         d.savedError = err
172     }
173 }
174
175 // next cuts off and returns the next full JSON value in d.d
176 // The next value is known to be an object or array, not a l
177 func (d *decodeState) next() []byte {
178     c := d.data[d.off]
179     item, rest, err := nextValue(d.data[d.off:], &d.next
180     if err != nil {
181         d.error(err)
182     }
183     d.off = len(d.data) - len(rest)
184
185     // Our scanner has seen the opening brace/bracket
186     // and thinks we're still in the middle of the objec
187     // invent a closing brace/bracket to get it out.
188     if c == '{' {
189         d.scan.step(&d.scan, '}')

```

```

190         } else {
191             d.scan.step(&d.scan, ']')
192         }
193
194         return item
195     }
196
197     // scanWhile processes bytes in d.data[d.off:] until it
198     // receives a scan code not equal to op.
199     // It updates d.off and returns the new scan code.
200     func (d *decodeState) scanWhile(op int) int {
201         var newOp int
202         for {
203             if d.off >= len(d.data) {
204                 newOp = d.scan.eof()
205                 d.off = len(d.data) + 1 // mark proc
206             } else {
207                 c := int(d.data[d.off])
208                 d.off++
209                 newOp = d.scan.step(&d.scan, c)
210             }
211             if newOp != op {
212                 break
213             }
214         }
215         return newOp
216     }
217
218     // value decodes a JSON value from d.data[d.off:] into the v
219     // it updates d.off to point past the decoded value.
220     func (d *decodeState) value(v reflect.Value) {
221         if !v.IsValid() {
222             _, rest, err := nextValue(d.data[d.off:], &
223             if err != nil {
224                 d.error(err)
225             }
226             d.off = len(d.data) - len(rest)
227
228             // d.scan thinks we're still at the beginnin
229             // Feed in an empty string - the shortest, s
230             // so that it knows we got to the end of the
231             if d.scan.redo {
232                 // rewind.
233                 d.scan.redo = false
234                 d.scan.step = stateBeginValue
235             }
236             d.scan.step(&d.scan, '')
237             d.scan.step(&d.scan, '')
238             return
239         }

```

```

240
241     switch op := d.scanWhile(scanSkipSpace); op {
242     default:
243         d.error(errPhase)
244
245     case scanBeginArray:
246         d.array(v)
247
248     case scanBeginObject:
249         d.object(v)
250
251     case scanBeginLiteral:
252         d.literal(v)
253     }
254 }
255
256 // indirect walks down v allocating pointers as needed,
257 // until it gets to a non-pointer.
258 // if it encounters an Unmarshaler, indirect stops and retur
259 // if decodingNull is true, indirect stops at the last point
260 func (d *decodeState) indirect(v reflect.Value, decodingNull
261     // If v is a named type and is addressable,
262     // start with its address, so that if the type has p
263     // we find them.
264     if v.Kind() != reflect.Ptr && v.Type().Name() != ""
265         v = v.Addr()
266     }
267     for {
268         var isUnmarshaler bool
269         if v.Type().NumMethod() > 0 {
270             // Remember that this is an unmarshala
271             // but wait to return it until after
272             // the pointer (if necessary).
273             _, isUnmarshaler = v.Interface().(Un
274         }
275
276         if iv := v; iv.Kind() == reflect.Interface &
277             v = iv.Elem()
278             continue
279         }
280
281         pv := v
282         if pv.Kind() != reflect.Ptr {
283             break
284         }
285
286         if pv.Elem().Kind() != reflect.Ptr && decodi
287             return nil, pv
288     }

```

```

289         if pv.IsNil() {
290             pv.Set(reflect.New(pv.Type().Elem()))
291         }
292         if isUnmarshaller {
293             // Using v.Interface().(Unmarshaller)
294             // here means that we have to use a
295             // as the struct field. We cannot u
296             // a pointer to a struct, because in
297             // v.Interface() is the value (x.f)
298             // This is an unfortunate consequenc
299             // An alternative would be to look u
300             // UnmarshalJSON method and return a
301             return v.Interface().(Unmarshaller),
302         }
303         v = pv.Elem()
304     }
305     return nil, v
306 }
307
308 // array consumes an array from d.data[d.off-1:], decoding i
309 // the first byte of the array ('[') has been read already.
310 func (d *decodeState) array(v reflect.Value) {
311     // Check for unmarshaller.
312     unmarshaller, pv := d.indirect(v, false)
313     if unmarshaller != nil {
314         d.off--
315         err := unmarshaller.UnmarshalJSON(d.next())
316         if err != nil {
317             d.error(err)
318         }
319         return
320     }
321     v = pv
322
323     // Check type of target.
324     switch v.Kind() {
325     default:
326         d.saveError(&UnmarshalTypeError{"array", v.T
327         d.off--
328         d.next()
329         return
330     case reflect.Interface:
331         // Decoding into nil interface? Switch to n
332         v.Set(reflect.ValueOf(d.arrayInterface()))
333         return
334     case reflect.Array:
335     case reflect.Slice:
336         break
337     }

```

```

338
339     i := 0
340     for {
341         // Look ahead for ] - can only happen on fir
342         op := d.scanWhile(scanSkipSpace)
343         if op == scanEndArray {
344             break
345         }
346
347         // Back up so d.value can have the byte we j
348         d.off--
349         d.scan.undo(op)
350
351         // Get element of array, growing if necessar
352         if v.Kind() == reflect.Slice {
353             // Grow slice if necessary
354             if i >= v.Cap() {
355                 newcap := v.Cap() + v.Cap()/
356                 if newcap < 4 {
357                     newcap = 4
358                 }
359                 newv := reflect.MakeSlice(v.
360                 reflect.Copy(newv, v)
361                 v.Set(newv)
362             }
363             if i >= v.Len() {
364                 v.SetLen(i + 1)
365             }
366         }
367
368         if i < v.Len() {
369             // Decode into element.
370             d.value(v.Index(i))
371         } else {
372             // Ran out of fixed array: skip.
373             d.value(reflect.Value{})
374         }
375         i++
376
377         // Next token must be , or ].
378         op = d.scanWhile(scanSkipSpace)
379         if op == scanEndArray {
380             break
381         }
382         if op != scanArrayValue {
383             d.error(errPhase)
384         }
385     }
386
387     if i < v.Len() {

```

```

388         if v.Kind() == reflect.Array {
389             // Array. Zero the rest.
390             z := reflect.Zero(v.Type().Elem())
391             for ; i < v.Len(); i++ {
392                 v.Index(i).Set(z)
393             }
394         } else {
395             v.SetLen(i)
396         }
397     }
398     if i == 0 && v.Kind() == reflect.Slice {
399         v.Set(reflect.MakeSlice(v.Type(), 0, 0))
400     }
401 }
402
403 // object consumes an object from d.data[d.off-1:], decoding
404 // the first byte of the object ('{') has been read already.
405 func (d *decodeState) object(v reflect.Value) {
406     // Check for unmarshaler.
407     unmarshaler, pv := d.indirect(v, false)
408     if unmarshaler != nil {
409         d.off--
410         err := unmarshaler.UnmarshalJSON(d.next())
411         if err != nil {
412             d.error(err)
413         }
414         return
415     }
416     v = pv
417
418     // Decoding into nil interface? Switch to non-refle
419     iv := v
420     if iv.Kind() == reflect.Interface {
421         iv.Set(reflect.ValueOf(d.objectInterface()))
422         return
423     }
424
425     // Check type of target: struct or map[string]T
426     var (
427         mv reflect.Value
428         sv reflect.Value
429     )
430     switch v.Kind() {
431     case reflect.Map:
432         // map must have string type
433         t := v.Type()
434         if t.Key() != reflect.TypeOf("") {
435             d.saveError(&UnmarshalTypeError{"obj
436             break

```

```

437         }
438         mv = v
439         if mv.IsNil() {
440             mv.Set(reflect.MakeMap(t))
441         }
442     case reflect.Struct:
443         sv = v
444     default:
445         d.saveError(&UnmarshalTypeError{"object", v.
446     }
447
448     if !mv.IsValid() && !sv.IsValid() {
449         d.off--
450         d.next() // skip over { } in input
451         return
452     }
453
454     var mapElem reflect.Value
455
456     for {
457         // Read opening " of string key or closing }
458         op := d.scanWhile(scanSkipSpace)
459         if op == scanEndObject {
460             // closing } - can only happen on fi
461             break
462         }
463         if op != scanBeginLiteral {
464             d.error(errPhase)
465         }
466
467         // Read string key.
468         start := d.off - 1
469         op = d.scanWhile(scanContinue)
470         item := d.data[start : d.off-1]
471         key, ok := unquote(item)
472         if !ok {
473             d.error(errPhase)
474         }
475
476         // Figure out field corresponding to key.
477         var subv reflect.Value
478         destring := false // whether the value is wr
479
480         if mv.IsValid() {
481             elemType := mv.Type().Elem()
482             if !mapElem.IsValid() {
483                 mapElem = reflect.New(elemTy
484             } else {
485                 mapElem.Set(reflect.Zero(ele

```

```

486         }
487         subv = mapElem
488     } else {
489         var f reflect.StructField
490         var ok bool
491         st := sv.Type()
492         for i := 0; i < sv.NumField(); i++ {
493             sf := st.Field(i)
494             tag := sf.Tag.Get("json")
495             if tag == "-" {
496                 // Pretend this field
497                 continue
498             }
499             if sf.Anonymous {
500                 // Pretend this field
501                 // so that we can do
502                 // these in a later
503                 continue
504             }
505             // First, tag match
506             tagName, _ := parseTag(tag)
507             if tagName == key {
508                 f = sf
509                 ok = true
510                 break // no better match
511             }
512             // Second, exact field name
513             if sf.Name == key {
514                 f = sf
515                 ok = true
516             }
517             // Third, case-insensitive match
518             // but only if a better match
519             if !ok && strings.EqualFold(
520                 f.Name, key) {
521                 f = sf
522                 ok = true
523             }
524         }
525         // Extract value; name must be exported
526         if ok {
527             if f.PkgPath != "" {
528                 d.saveError(&Unmarshaled)
529             } else {
530                 subv = sv.FieldByIndex(f.Index)
531             }
532             _, opts := parseTag(f.Tag.Get("json"))
533             destr := destr + f.Name + opts.Contains("string")
534         }
535     }

```

```

536
537 // Read : before value.
538 if op == scanSkipSpace {
539     op = d.scanWhile(scanSkipSpace)
540 }
541 if op != scanObjectKey {
542     d.error(errPhase)
543 }
544
545 // Read value.
546 if destring {
547     d.value(reflect.ValueOf(&d.tempstr))
548     d.literalStore([]byte(d.tempstr), su
549 } else {
550     d.value(subv)
551 }
552 // Write value back to map;
553 // if using struct, subv points into struct
554 if mv.IsValid() {
555     mv.SetMapIndex(reflect.ValueOf(key),
556 }
557
558 // Next token must be , or }.
559 op = d.scanWhile(scanSkipSpace)
560 if op == scanEndObject {
561     break
562 }
563 if op != scanObjectValue {
564     d.error(errPhase)
565 }
566     }
567 }
568
569 // literal consumes a literal from d.data[d.off-1:], decodin
570 // The first byte of the literal has been read already
571 // (that's how the caller knows it's a literal).
572 func (d *decodeState) literal(v reflect.Value) {
573     // All bytes inside literal return scanContinue op c
574     start := d.off - 1
575     op := d.scanWhile(scanContinue)
576
577     // Scan read one byte too far; back up.
578     d.off--
579     d.scan.undo(op)
580
581     d.literalStore(d.data[start:d.off], v, false)
582 }
583
584 // literalStore decodes a literal stored in item into v.

```

```

585 //
586 // fromQuoted indicates whether this literal came from unwra
587 // string from the ",string" struct tag option. this is used
588 // produce more helpful error messages.
589 func (d *decodeState) literalStore(item []byte, v reflect.Va
590     // Check for unmarshaler.
591     wantptr := item[0] == 'n' // null
592     unmarshaler, pv := d.indirect(v, wantptr)
593     if unmarshaler != nil {
594         err := unmarshaler.UnmarshalJSON(item)
595         if err != nil {
596             d.error(err)
597         }
598     }
599     return
600     }
601     v = pv
602
603     switch c := item[0]; c {
604     case 'n': // null
605         switch v.Kind() {
606         default:
607             d.saveError(&UnmarshalTypeError{"nul
608             case reflect.Interface, reflect.Ptr, reflect
609                 v.Set(reflect.Zero(v.Type()))
610         }
611     case 't', 'f': // true, false
612         value := c == 't'
613         switch v.Kind() {
614         default:
615             if fromQuoted {
616                 d.saveError(fmt.Errorf("json
617             } else {
618                 d.saveError(&UnmarshalTypeEr
619             }
620         case reflect.Bool:
621             v.SetBool(value)
622         case reflect.Interface:
623             v.Set(reflect.ValueOf(value))
624         }
625     case '"': // string
626         s, ok := unquoteBytes(item)
627         if !ok {
628             if fromQuoted {
629                 d.error(fmt.Errorf("json: in
630             } else {
631                 d.error(errPhase)
632             }
633         }

```

```

634     }
635     switch v.Kind() {
636     default:
637         d.saveError(&UnmarshalTypeError{"str
638 case reflect.Slice:
639         if v.Type() != byteSliceType {
640             d.saveError(&UnmarshalTypeEr
641                 break
642         }
643         b := make([]byte, base64.StdEncoding
644         n, err := base64.StdEncoding.Decode(
645         if err != nil {
646             d.saveError(err)
647                 break
648         }
649         v.Set(reflect.ValueOf(b[0:n]))
650 case reflect.String:
651         v.SetString(string(s))
652 case reflect.Interface:
653         v.Set(reflect.ValueOf(string(s)))
654     }
655
656     default: // number
657         if c != '-' && (c < '0' || c > '9') {
658             if fromQuoted {
659                 d.error(fmt.Errorf("json: in
660             } else {
661                 d.error(errPhase)
662             }
663         }
664         s := string(item)
665         switch v.Kind() {
666         default:
667             if fromQuoted {
668                 d.error(fmt.Errorf("json: in
669             } else {
670                 d.error(&UnmarshalTypeError{
671             }
672         case reflect.Interface:
673             n, err := strconv.ParseFloat(s, 64)
674             if err != nil {
675                 d.saveError(&UnmarshalTypeEr
676                 break
677             }
678             v.Set(reflect.ValueOf(n))
679
680         case reflect.Int, reflect.Int8, reflect.Int1
681             n, err := strconv.ParseInt(s, 10, 64
682             if err != nil || v.OverflowInt(n) {
683                 d.saveError(&UnmarshalTypeEr

```

```

684             break
685         }
686         v.SetInt(n)
687
688         case reflect.Uint, reflect.Uint8, reflect.Ui
689             n, err := strconv.ParseUint(s, 10, 6
690             if err != nil || v.OverflowUint(n) {
691                 d.saveError(&UnmarshalTypeError
692                 break
693             }
694             v.SetUint(n)
695
696         case reflect.Float32, reflect.Float64:
697             n, err := strconv.ParseFloat(s, v.Ty
698             if err != nil || v.OverflowFloat(n)
699                 d.saveError(&UnmarshalTypeError
700                 break
701             }
702             v.SetFloat(n)
703     }
704 }
705 }
706
707 // The xxxInterface routines build up a value to be stored
708 // in an empty interface. They are not strictly necessary,
709 // but they avoid the weight of reflection in this common ca
710
711 // valueInterface is like value but returns interface{}
712 func (d *decodeState) valueInterface() interface{} {
713     switch d.scanWhile(scanSkipSpace) {
714     default:
715         d.error(errPhase)
716     case scanBeginArray:
717         return d.arrayInterface()
718     case scanBeginObject:
719         return d.objectInterface()
720     case scanBeginLiteral:
721         return d.literalInterface()
722     }
723     panic("unreachable")
724 }
725
726 // arrayInterface is like array but returns []interface{}.
727 func (d *decodeState) arrayInterface() []interface{} {
728     var v []interface{}
729     for {
730         // Look ahead for ] - can only happen on fir
731         op := d.scanWhile(scanSkipSpace)
732         if op == scanEndArray {

```

```

733             break
734         }
735
736         // Back up so d.value can have the byte we j
737         d.off--
738         d.scan.undo(op)
739
740         v = append(v, d.valueInterface())
741
742         // Next token must be , or ].
743         op = d.scanWhile(scanSkipSpace)
744         if op == scanEndArray {
745             break
746         }
747         if op != scanArrayValue {
748             d.error(errPhase)
749         }
750     }
751     return v
752 }
753
754 // objectInterface is like object but returns map[string]int
755 func (d *decodeState) objectInterface() map[string]interface
756     m := make(map[string]interface{})
757     for {
758         // Read opening " of string key or closing }
759         op := d.scanWhile(scanSkipSpace)
760         if op == scanEndObject {
761             // closing } - can only happen on fi
762             break
763         }
764         if op != scanBeginLiteral {
765             d.error(errPhase)
766         }
767
768         // Read string key.
769         start := d.off - 1
770         op = d.scanWhile(scanContinue)
771         item := d.data[start : d.off-1]
772         key, ok := unquote(item)
773         if !ok {
774             d.error(errPhase)
775         }
776
777         // Read : before value.
778         if op == scanSkipSpace {
779             op = d.scanWhile(scanSkipSpace)
780         }
781         if op != scanObjectKey {

```

```

782             d.error(errPhase)
783         }
784
785         // Read value.
786         m[key] = d.valueInterface()
787
788         // Next token must be , or }.
789         op = d.scanWhile(scanSkipSpace)
790         if op == scanEndObject {
791             break
792         }
793         if op != scanObjectValue {
794             d.error(errPhase)
795         }
796     }
797     return m
798 }
799
800 // literalInterface is like literal but returns an interface
801 func (d *decodeState) literalInterface() interface{} {
802     // All bytes inside literal return scanContinue op c
803     start := d.off - 1
804     op := d.scanWhile(scanContinue)
805
806     // Scan read one byte too far; back up.
807     d.off--
808     d.scan.undo(op)
809     item := d.data[start:d.off]
810
811     switch c := item[0]; c {
812     case 'n': // null
813         return nil
814
815     case 't', 'f': // true, false
816         return c == 't'
817
818     case '"': // string
819         s, ok := unquote(item)
820         if !ok {
821             d.error(errPhase)
822         }
823         return s
824
825     default: // number
826         if c != '-' && (c < '0' || c > '9') {
827             d.error(errPhase)
828         }
829         n, err := strconv.ParseFloat(string(item), 6)
830         if err != nil {
831             d.saveError(&UnmarshalTypeError{"num

```

```

832         }
833         return n
834     }
835     panic("unreachable")
836 }
837
838 // getu4 decodes \uXXXX from the beginning of s, returning t
839 // or it returns -1.
840 func getu4(s []byte) rune {
841     if len(s) < 6 || s[0] != '\\' || s[1] != 'u' {
842         return -1
843     }
844     r, err := strconv.ParseUint(string(s[2:6]), 16, 64)
845     if err != nil {
846         return -1
847     }
848     return rune(r)
849 }
850
851 // unquote converts a quoted JSON string literal s into an a
852 // The rules are different than for Go, so cannot use strconv
853 func unquote(s []byte) (t string, ok bool) {
854     s, ok = unquoteBytes(s)
855     t = string(s)
856     return
857 }
858
859 func unquoteBytes(s []byte) (t []byte, ok bool) {
860     if len(s) < 2 || s[0] != '"' || s[len(s)-1] != '"' {
861         return
862     }
863     s = s[1 : len(s)-1]
864
865     // Check for unusual characters. If there are none,
866     // then no unquoting is needed, so return a slice of
867     // original bytes.
868     r := 0
869     for r < len(s) {
870         c := s[r]
871         if c == '\\' || c == '"' || c < ' ' {
872             break
873         }
874         if c < utf8.RuneSelf {
875             r++
876             continue
877         }
878         rr, size := utf8.DecodeRune(s[r:])
879         if rr == utf8.RuneError && size == 1 {
880             break

```

```

881         }
882         r += size
883     }
884     if r == len(s) {
885         return s, true
886     }
887
888     b := make([]byte, len(s)+2*utf8.UTFMax)
889     w := copy(b, s[0:r])
890     for r < len(s) {
891         // Out of room? Can only happen if s is full
892         // malformed UTF-8 and we're replacing each
893         // byte with RuneError.
894         if w >= len(b)-2*utf8.UTFMax {
895             nb := make([]byte, (len(b)+utf8.UTFM
896             copy(nb, b[0:w])
897             b = nb
898         }
899         switch c := s[r]; {
900         case c == '\\':
901             r++
902             if r >= len(s) {
903                 return
904             }
905             switch s[r] {
906             default:
907                 return
908             case '"', '\\', '/', '\':
909                 b[w] = s[r]
910                 r++
911                 w++
912             case 'b':
913                 b[w] = '\b'
914                 r++
915                 w++
916             case 'f':
917                 b[w] = '\f'
918                 r++
919                 w++
920             case 'n':
921                 b[w] = '\n'
922                 r++
923                 w++
924             case 'r':
925                 b[w] = '\r'
926                 r++
927                 w++
928             case 't':
929                 b[w] = '\t'

```

```

930         r++
931         w++
932         case 'u':
933             r--
934             rr := getu4(s[r:])
935             if rr < 0 {
936                 return
937             }
938             r += 6
939             if utf16.IsSurrogate(rr) {
940                 rr1 := getu4(s[r:])
941                 if dec := utf16.Deco
942                     // A valid p
943                     r += 6
944                     w += utf8.En
945                     break
946                 }
947                 // Invalid surrogate
948                 rr = unicode.Replace
949             }
950             w += utf8.EncodeRune(b[w:],
951         }
952
953         // Quote, control characters are invalid.
954         case c == '"', c < ' ':
955             return
956
957         // ASCII
958         case c < utf8.RuneSelf:
959             b[w] = c
960             r++
961             w++
962
963         // Coerce to well-formed UTF-8.
964         default:
965             rr, size := utf8.DecodeRune(s[r:])
966             r += size
967             w += utf8.EncodeRune(b[w:], rr)
968         }
969     }
970     return b[0:w], true
971 }
972
973 // The following is issue 3069.
974
975 // BUG(rsc): This package ignores anonymous (embedded) struc
976 // during encoding and decoding. A future version may assign
977 // to them. To force an anonymous field to be ignored in all
978 // versions of this package, use an explicit `json:"-"` tag
979 // definition.

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/json/encode.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package json implements encoding and decoding of JSON obj
6 // RFC 4627.
7 //
8 // See "JSON and Go" for an introduction to this package:
9 // http://golang.org/doc/articles/json\_and\_go.html
10 package json
11
12 import (
13     "bytes"
14     "encoding/base64"
15     "math"
16     "reflect"
17     "runtime"
18     "sort"
19     "strconv"
20     "strings"
21     "sync"
22     "unicode"
23     "unicode/utf8"
24 )
25
26 // Marshal returns the JSON encoding of v.
27 //
28 // Marshal traverses the value v recursively.
29 // If an encountered value implements the Marshaler interface
30 // and is not a nil pointer, Marshal calls its MarshalJSON method
31 // to produce JSON. The nil pointer exception is not strict
32 // but mimics a similar, necessary exception in the behavior
33 // of UnmarshalJSON.
34 //
35 // Otherwise, Marshal uses the following type-dependent default
36 // encodings:
37 // Boolean values encode as JSON booleans.
38 //
39 // Floating point and integer values encode as JSON numbers.
40 //
41 // String values encode as JSON strings, with each invalid U
```

```

42 // replaced by the encoding of the Unicode replacement chara
43 // The angle brackets "<" and ">" are escaped to "\u003c" an
44 // to keep some browsers from misinterpreting JSON output as
45 //
46 // Array and slice values encode as JSON arrays, except that
47 // []byte encodes as a base64-encoded string, and a nil slic
48 // encodes as the null JSON object.
49 //
50 // Struct values encode as JSON objects. Each exported struc
51 // becomes a member of the object unless
52 //   - the field's tag is "-", or
53 //   - the field is empty and its tag specifies the "omitempty"
54 // The empty values are false, 0, any
55 // nil pointer or interface value, and any array, slice, map
56 // length zero. The object's default key string is the struc
57 // but can be specified in the struct field's tag value. The
58 // struct field's tag value is the key name, followed by an
59 // and options. Examples:
60 //
61 //   // Field is ignored by this package.
62 //   Field int `json:"- "`
63 //
64 //   // Field appears in JSON as key "myName".
65 //   Field int `json:"myName"`
66 //
67 //   // Field appears in JSON as key "myName" and
68 //   // the field is omitted from the object if its value is
69 //   // as defined above.
70 //   Field int `json:"myName,omitempty"`
71 //
72 //   // Field appears in JSON as key "Field" (the default),
73 //   // the field is skipped if empty.
74 //   // Note the leading comma.
75 //   Field int `json:",omitempty"`
76 //
77 // The "string" option signals that a field is stored as JSO
78 // JSON-encoded string. This extra level of encoding is som
79 // used when communicating with JavaScript programs:
80 //
81 //   Int64String int64 `json:",string"`
82 //
83 // The key name will be used if it's a non-empty string cons
84 // only Unicode letters, digits, dollar signs, percent signs
85 // underscores and slashes.
86 //
87 // Map values encode as JSON objects.
88 // The map's key type must be string; the object keys are us
89 // as map keys.
90 //
91 // Pointer values encode as the value pointed to.

```

```

92 // A nil pointer encodes as the null JSON object.
93 //
94 // Interface values encode as the value contained in the int
95 // A nil interface value encodes as the null JSON object.
96 //
97 // Channel, complex, and function values cannot be encoded i
98 // Attempting to encode such a value causes Marshal to retur
99 // an InvalidTypeError.
100 //
101 // JSON cannot represent cyclic data structures and Marshal
102 // handle them. Passing cyclic structures to Marshal will r
103 // an infinite recursion.
104 //
105 func Marshal(v interface{}) ([]byte, error) {
106     e := &encodeState{}
107     err := e.marshal(v)
108     if err != nil {
109         return nil, err
110     }
111     return e.Bytes(), nil
112 }
113
114 // MarshalIndent is like Marshal but applies Indent to forma
115 func MarshalIndent(v interface{}, prefix, indent string) ([]
116     b, err := Marshal(v)
117     if err != nil {
118         return nil, err
119     }
120     var buf bytes.Buffer
121     err = Indent(&buf, b, prefix, indent)
122     if err != nil {
123         return nil, err
124     }
125     return buf.Bytes(), nil
126 }
127
128 // HTML Escape appends to dst the JSON-encoded src with <, >,
129 // characters inside string literals changed to \u003c, \u00
130 // so that the JSON will be safe to embed inside HTML <scrip
131 // For historical reasons, web browsers don't honor standard
132 // escaping within <script> tags, so an alternative JSON enc
133 // be used.
134 func HTML Escape(dst *bytes.Buffer, src []byte) {
135     // < > & can only appear in string literals,
136     // so just scan the string one byte at a time.
137     start := 0
138     for i, c := range src {
139         if c == '<' || c == '>' || c == '&' {
140             if start < i {

```

```

141             dst.Write(src[start:i])
142         }
143         dst.WriteString(`\u00`)
144         dst.WriteByte(hex[c>>4])
145         dst.WriteByte(hex[c&0xF])
146         start = i + 1
147     }
148 }
149 if start < len(src) {
150     dst.Write(src[start:])
151 }
152 }
153
154 // Marshaler is the interface implemented by objects that
155 // can marshal themselves into valid JSON.
156 type Marshaler interface {
157     MarshalJSON() ([]byte, error)
158 }
159
160 type UnsupportedTypeError struct {
161     Type reflect.Type
162 }
163
164 func (e *UnsupportedTypeError) Error() string {
165     return "json: unsupported type: " + e.Type.String()
166 }
167
168 type UnsupportedValueError struct {
169     Value reflect.Value
170     Str    string
171 }
172
173 func (e *UnsupportedValueError) Error() string {
174     return "json: unsupported value: " + e.Str
175 }
176
177 type InvalidUTF8Error struct {
178     S string
179 }
180
181 func (e *InvalidUTF8Error) Error() string {
182     return "json: invalid UTF-8 in string: " + strconv.Q
183 }
184
185 type MarshalerError struct {
186     Type reflect.Type
187     Err  error
188 }
189

```

```

190 func (e *MarshalerError) Error() string {
191     return "json: error calling MarshalJSON for type " +
192 }
193
194 var hex = "0123456789abcdef"
195
196 // An encodeState encodes JSON into a bytes.Buffer.
197 type encodeState struct {
198     bytes.Buffer // accumulated output
199     scratch      [64]byte
200 }
201
202 func (e *encodeState) marshal(v interface{}) (err error) {
203     defer func() {
204         if r := recover(); r != nil {
205             if _, ok := r.(runtime.Error); ok {
206                 panic(r)
207             }
208             err = r.(error)
209         }
210     }()
211     e.reflectValue(reflect.ValueOf(v))
212     return nil
213 }
214
215 func (e *encodeState) error(err error) {
216     panic(err)
217 }
218
219 var byteSliceType = reflect.TypeOf([]byte(nil))
220
221 func isEmptyValue(v reflect.Value) bool {
222     switch v.Kind() {
223     case reflect.Array, reflect.Map, reflect.Slice, refl
224         return v.Len() == 0
225     case reflect.Bool:
226         return !v.Bool()
227     case reflect.Int, reflect.Int8, reflect.Int16, refle
228         return v.Int() == 0
229     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
230         return v.Uint() == 0
231     case reflect.Float32, reflect.Float64:
232         return v.Float() == 0
233     case reflect.Interface, reflect.Ptr:
234         return v.IsNil()
235     }
236     return false
237 }
238
239 func (e *encodeState) reflectValue(v reflect.Value) {

```

```

240         e.reflectValueQuoted(v, false)
241     }
242
243     // reflectValueQuoted writes the value in v to the output.
244     // If quoted is true, the serialization is wrapped in a JSON
245     func (e *encodeState) reflectValueQuoted(v reflect.Value, qu
246         if !v.IsValid() {
247             e.WriteString("null")
248             return
249         }
250
251         m, ok := v.Interface().(Marshaler)
252         if !ok {
253             // T doesn't match the interface. Check agai
254             if v.Kind() != reflect.Ptr && v.CanAddr() {
255                 m, ok = v.Addr().Interface().(Marsha
256                 if ok {
257                     v = v.Addr()
258                 }
259             }
260         }
261         if ok && (v.Kind() != reflect.Ptr || !v.IsNil()) {
262             b, err := m.MarshalJSON()
263             if err == nil {
264                 // copy JSON into buffer, checking v
265                 err = compact(&e.Buffer, b, true)
266             }
267             if err != nil {
268                 e.error(&MarshalerError{v.Type(), er
269             }
270             return
271         }
272
273         writeString := (*encodeState).WriteString
274         if quoted {
275             writeString = (*encodeState).string
276         }
277
278         switch v.Kind() {
279         case reflect.Bool:
280             x := v.Bool()
281             if x {
282                 writeString(e, "true")
283             } else {
284                 writeString(e, "false")
285             }
286
287         case reflect.Int, reflect.Int8, reflect.Int16, refle
288             b := strconv.AppendInt(e.scratch[:0], v.Int(

```

```

289         if quoted {
290             writeString(e, string(b))
291         } else {
292             e.Write(b)
293         }
294     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
295         b := strconv.AppendUint(e.scratch[:0], v.Uin
296         if quoted {
297             writeString(e, string(b))
298         } else {
299             e.Write(b)
300         }
301     case reflect.Float32, reflect.Float64:
302         f := v.Float()
303         if math.IsInf(f, 0) || math.IsNaN(f) {
304             e.error(&UnsupportedValueError{v, st
305         }
306         b := strconv.AppendFloat(e.scratch[:0], f, '
307         if quoted {
308             writeString(e, string(b))
309         } else {
310             e.Write(b)
311         }
312     case reflect.String:
313         if quoted {
314             sb, err := Marshal(v.String())
315             if err != nil {
316                 e.error(err)
317             }
318             e.string(string(sb))
319         } else {
320             e.string(v.String())
321         }
322
323     case reflect.Struct:
324         e.WriteByte('{')
325         first := true
326         for _, ef := range encodeFields(v.Type()) {
327             fieldValue := v.Field(ef.i)
328             if ef.omitEmpty && isEmptyValue(fiel
329                 continue
330             }
331             if first {
332                 first = false
333             } else {
334                 e.WriteByte(',')
335             }
336             e.string(ef.tag)
337             e.WriteByte(':')

```

```

338         e.reflectValueQuoted(fieldValue, ef.
339     }
340     e.WriteByte('}')
341
342     case reflect.Map:
343         if v.Type().Key().Kind() != reflect.String {
344             e.error(&UnsupportedTypeError{v.Type
345         }
346         if v.IsNil() {
347             e.WriteString("null")
348             break
349         }
350         e.WriteByte('{')
351         var sv stringValues = v.MapKeys()
352         sort.Sort(sv)
353         for i, k := range sv {
354             if i > 0 {
355                 e.WriteByte(',')
356             }
357             e.string(k.String())
358             e.WriteByte(':')
359             e.reflectValue(v.MapIndex(k))
360         }
361         e.WriteByte('}')
362
363     case reflect.Slice:
364         if v.IsNil() {
365             e.WriteString("null")
366             break
367         }
368         if v.Type().Elem().Kind() == reflect.Uint8 {
369             // Byte slices get special treatment
370             s := v.Bytes()
371             e.WriteByte('"')
372             if len(s) < 1024 {
373                 // for small buffers, using
374                 dst := make([]byte, base64.S
375                 base64.StdEncoding.Encode(ds
376                 e.Write(dst)
377             } else {
378                 // for large buffers, avoid
379                 // buffer space.
380                 enc := base64.NewEncoder(bas
381                 enc.Write(s)
382                 enc.Close()
383             }
384             e.WriteByte('"')
385             break
386         }
387         // Slices can be marshalled as nil, but othe

```

```

388         // as arrays.
389         fallthrough
390     case reflect.Array:
391         e.WriteByte('[')
392         n := v.Len()
393         for i := 0; i < n; i++ {
394             if i > 0 {
395                 e.WriteByte(',')
396             }
397             e.reflectValue(v.Index(i))
398         }
399         e.WriteByte(']')
400
401     case reflect.Interface, reflect.Ptr:
402         if v.IsNil() {
403             e.WriteString("null")
404             return
405         }
406         e.reflectValue(v.Elem())
407
408     default:
409         e.error(&UnsupportedTypeError{v.Type()})
410     }
411     return
412 }
413
414 func isValidTag(s string) bool {
415     if s == "" {
416         return false
417     }
418     for _, c := range s {
419         switch {
420         case strings.ContainsRune("!#$%&()*+-./:<=>?"
421             // Backslash and quote chars are res
422             // otherwise any punctuation chars a
423             // in a tag name.
424         default:
425             if !unicode.IsLetter(c) && !unicode.
426                 return false
427         }
428     }
429     return true
430 }
431 }
432
433 // stringValues is a slice of reflect.Value holding *reflect
434 // It implements the methods to sort by string.
435 type stringValues []reflect.Value
436

```

```

437 func (sv stringValues) Len() int           { return len(sv)
438 func (sv stringValues) Swap(i, j int)      { sv[i], sv[j] =
439 func (sv stringValues) Less(i, j int) bool { return sv.get(i
440 func (sv stringValues) get(i int) string    { return sv[i].St
441
442 func (e *encodeState) string(s string) (int, error) {
443     len0 := e.Len()
444     e.WriteByte('')
445     start := 0
446     for i := 0; i < len(s); {
447         if b := s[i]; b < utf8.RuneSelf {
448             if 0x20 <= b && b != '\\\ ' && b != ' '
449                 i++
450                 continue
451         }
452         if start < i {
453             e.WriteString(s[start:i])
454         }
455         switch b {
456         case '\\\ ', '\":
457             e.WriteByte('\\\\\ ')
458             e.WriteByte(b)
459         case '\n':
460             e.WriteByte('\\\\\n')
461             e.WriteByte('n')
462         case '\r':
463             e.WriteByte('\\\\\r')
464             e.WriteByte('r')
465         default:
466             // This encodes bytes < 0x20
467             // as well as < and >. The l
468             // can lead to security hole
469             // are rendered into JSON an
470             e.WriteString(`\u00`)
471             e.WriteByte(hex[b>>4])
472             e.WriteByte(hex[b&0xF])
473         }
474         i++
475         start = i
476         continue
477     }
478     c, size := utf8.DecodeRuneInString(s[i:])
479     if c == utf8.RuneError && size == 1 {
480         e.error(&InvalidUTF8Error{s})
481     }
482     i += size
483 }
484 if start < len(s) {
485     e.WriteString(s[start:])

```

```

486     }
487     e.WriteByte('')
488     return e.Len() - len0, nil
489 }
490
491 // encodeField contains information about how to encode a fi
492 // struct.
493 type encodeField struct {
494     i      int // field index in struct
495     tag    string
496     quoted bool
497     omitEmpty bool
498 }
499
500 var (
501     typeCacheLock sync.RWMutex
502     encodeFieldsCache = make(map[reflect.Type][]encodeFi
503 )
504
505 // encodeFields returns a slice of encodeField for a given
506 // struct type.
507 func encodeFields(t reflect.Type) []encodeField {
508     typeCacheLock.RLock()
509     fs, ok := encodeFieldsCache[t]
510     typeCacheLock.RUnlock()
511     if ok {
512         return fs
513     }
514
515     typeCacheLock.Lock()
516     defer typeCacheLock.Unlock()
517     fs, ok = encodeFieldsCache[t]
518     if ok {
519         return fs
520     }
521
522     v := reflect.Zero(t)
523     n := v.NumField()
524     for i := 0; i < n; i++ {
525         f := t.Field(i)
526         if f.PkgPath != "" {
527             continue
528         }
529         if f.Anonymous {
530             // We want to do a better job with t
531             // so for now pretend they don't exi
532             continue
533         }
534         var ef encodeField
535         ef.i = i

```

```
536         ef.tag = f.Name
537
538         tv := f.Tag.Get("json")
539         if tv != "" {
540             if tv == "-" {
541                 continue
542             }
543             name, opts := parseTag(tv)
544             if isValidTag(name) {
545                 ef.tag = name
546             }
547             ef.omitEmpty = opts.Contains("omitempty")
548             ef.quoted = opts.Contains("string")
549         }
550         fs = append(fs, ef)
551     }
552     encodeFieldsCache[t] = fs
553     return fs
554 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/json/indent.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package json
6
7 import "bytes"
8
9 // Compact appends to dst the JSON-encoded src with
10 // insignificant space characters elided.
11 func Compact(dst *bytes.Buffer, src []byte) error {
12     return compact(dst, src, false)
13 }
14
15 func compact(dst *bytes.Buffer, src []byte, escape bool) error {
16     origLen := dst.Len()
17     var scan scanner
18     scan.reset()
19     start := 0
20     for i, c := range src {
21         if escape && (c == '<' || c == '>' || c == ' ' ||
22             c == '\n' || c == '\r' || c == '\t') {
23             if start < i {
24                 dst.Write(src[start:i])
25             }
26             dst.WriteString(`\u00`)
27             dst.WriteByte(hex[c>>4])
28             dst.WriteByte(hex[c&0xF])
29             start = i + 1
30         }
31         v := scan.step(&scan, int(c))
32         if v >= scanSkipSpace {
33             if v == scanError {
34                 break
35             }
36             if start < i {
37                 dst.Write(src[start:i])
38             }
39             start = i + 1
40         }
41     }
42     if scan.eof() == scanError {
```

```

42         dst.Truncate(origLen)
43         return scan.err
44     }
45     if start < len(src) {
46         dst.Write(src[start:])
47     }
48     return nil
49 }
50
51 func newline(dst *bytes.Buffer, prefix, indent string, depth
52     dst.WriteByte('\n')
53     dst.WriteString(prefix)
54     for i := 0; i < depth; i++ {
55         dst.WriteString(indent)
56     }
57 }
58
59 // Indent appends to dst an indented form of the JSON-encode
60 // Each element in a JSON object or array begins on a new,
61 // indented line beginning with prefix followed by one or mo
62 // copies of indent according to the indentation nesting.
63 // The data appended to dst has no trailing newline, to make
64 // to embed inside other formatted JSON data.
65 func Indent(dst *bytes.Buffer, src []byte, prefix, indent st
66     origLen := dst.Len()
67     var scan scanner
68     scan.reset()
69     needIndent := false
70     depth := 0
71     for _, c := range src {
72         scan.bytes++
73         v := scan.step(&scan, int(c))
74         if v == scanSkipSpace {
75             continue
76         }
77         if v == scanError {
78             break
79         }
80         if needIndent && v != scanEndObject && v !=
81             needIndent = false
82             depth++
83             newline(dst, prefix, indent, depth)
84     }
85
86     // Emit semantically uninteresting bytes
87     // (in particular, punctuation in strings) u
88     if v == scanContinue {
89         dst.WriteByte(c)
90         continue
91     }

```

```

92
93     // Add spacing around real punctuation.
94     switch c {
95     case '{', '[':
96         // delay indent so that empty object
97         needIndent = true
98         dst.WriteByte(c)
99
100    case ',':
101        dst.WriteByte(c)
102        newline(dst, prefix, indent, depth)
103
104    case ':':
105        dst.WriteByte(c)
106        dst.WriteByte(' ')
107
108    case '}', ']':
109        if needIndent {
110            // suppress indent in empty
111            needIndent = false
112        } else {
113            depth--
114            newline(dst, prefix, indent,
115                )
116            dst.WriteByte(c)
117
118        default:
119            dst.WriteByte(c)
120        }
121    }
122    if scan.eof() == scanError {
123        dst.Truncate(origLen)
124        return scan.err
125    }
126    return nil
127 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/json/scanner.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package json
6
7 // JSON value parser state machine.
8 // Just about at the limit of what is reasonable to write by
9 // Some parts are a bit tedious, but overall it nicely facto
10 // otherwise common code from the multiple scanning function
11 // in this package (Compact, Indent, checkValid, nextValue,
12 //
13 // This file starts with two simple examples using the scann
14 // before diving into the scanner itself.
15
16 import "strconv"
17
18 // checkValid verifies that data is valid JSON-encoded data.
19 // scan is passed in for use by checkValid to avoid an alloc
20 func checkValid(data []byte, scan *scanner) error {
21     scan.reset()
22     for _, c := range data {
23         scan.bytes++
24         if scan.step(scan, int(c)) == scanError {
25             return scan.err
26         }
27     }
28     if scan.eof() == scanError {
29         return scan.err
30     }
31     return nil
32 }
33
34 // nextValue splits data after the next whole JSON value,
35 // returning that value and the bytes that follow it as sepa
36 // scan is passed in for use by nextValue to avoid an alloca
37 func nextValue(data []byte, scan *scanner) (value, rest []by
38     scan.reset()
39     for i, c := range data {
40         v := scan.step(scan, int(c))
41         if v >= scanEnd {
```

```

42             switch v {
43             case scanError:
44                 return nil, nil, scan.err
45             case scanEnd:
46                 return data[0:i], data[i:],
47             }
48         }
49     }
50     if scan.eof() == scanError {
51         return nil, nil, scan.err
52     }
53     return data, nil, nil
54 }
55
56 // A SyntaxError is a description of a JSON syntax error.
57 type SyntaxError struct {
58     msg     string // description of error
59     Offset int64  // error occurred after reading Offset
60 }
61
62 func (e *SyntaxError) Error() string { return e.msg }
63
64 // A scanner is a JSON scanning state machine.
65 // Callers call scan.reset() and then pass bytes in one at a
66 // by calling scan.step(&scan, c) for each byte.
67 // The return value, referred to as an opcode, tells the
68 // caller about significant parsing events like beginning
69 // and ending literals, objects, and arrays, so that the
70 // caller can follow along if it wishes.
71 // The return value scanEnd indicates that a single top-level
72 // JSON value has been completed, *before* the byte that
73 // just got passed in. (The indication must be delayed in c
74 // to recognize the end of numbers: is 123 a whole value or
75 // the beginning of 12345e+6?).
76 type scanner struct {
77     // The step is a func to be called to execute the ne
78     // Also tried using an integer constant and a single
79     // with a switch, but using the func directly was 10
80     // on a 64-bit Mac Mini, and it's nicer to read.
81     step func(*scanner, int) int
82
83     // Reached end of top-level value.
84     endTop bool
85
86     // Stack of what we're in the middle of - array valu
87     parseState []int
88
89     // Error that happened, if any.
90     err error
91

```

```

92         // 1-byte redo (see undo method)
93         redo         bool
94         redoCode    int
95         redoState  func(*scanner, int) int
96
97         // total bytes consumed, updated by decoder.Decode
98         bytes       int64
99     }
100
101 // These values are returned by the state transition function
102 // assigned to scanner.state and the method scanner.eof.
103 // They give details about the current state of the scan that
104 // callers might be interested to know about.
105 // It is okay to ignore the return value of any particular
106 // call to scanner.state: if one call returns scanError,
107 // every subsequent call will return scanError too.
108 const (
109     // Continue.
110     scanContinue      = iota // uninteresting byte
111     scanBeginLiteral // end implied by next result
112     scanBeginObject  // begin object
113     scanObjectKey    // just finished object key
114     scanObjectValue  // just finished non-last object value
115     scanEndObject    // end object (implies scanContinue)
116     scanBeginArray   // begin array
117     scanArrayValue   // just finished array value
118     scanEndArray     // end array (implies scanContinue)
119     scanSkipSpace    // space byte; can skip; known
120
121     // Stop.
122     scanEnd // top-level value ended *before* this byte
123     scanError // hit an error, scanner.err.
124 )
125
126 // These values are stored in the parseState stack.
127 // They give the current state of a composite value
128 // being scanned. If the parser is inside a nested value
129 // the parseState describes the nested state, outermost at end.
130 const (
131     parseObjectKey = iota // parsing object key (before value)
132     parseObjectValue // parsing object value (after key)
133     parseArrayValue // parsing array value
134 )
135
136 // reset prepares the scanner for use.
137 // It must be called before calling s.step.
138 func (s *scanner) reset() {
139     s.step = stateBeginValue
140     s.parseState = s.parseState[0:0]

```

```

141         s.err = nil
142         s.redo = false
143         s.endTop = false
144     }
145
146     // eof tells the scanner that the end of input has been reac
147     // It returns a scan status just as s.step does.
148     func (s *scanner) eof() int {
149         if s.err != nil {
150             return scanError
151         }
152         if s.endTop {
153             return scanEnd
154         }
155         s.step(s, ' ')
156         if s.endTop {
157             return scanEnd
158         }
159         if s.err == nil {
160             s.err = &SyntaxError{"unexpected end of JSON
161         }
162         return scanError
163     }
164
165     // pushParseState pushes a new parse state p onto the parse
166     func (s *scanner) pushParseState(p int) {
167         s.parseState = append(s.parseState, p)
168     }
169
170     // popParseState pops a parse state (already obtained) off t
171     // and updates s.step accordingly.
172     func (s *scanner) popParseState() {
173         n := len(s.parseState) - 1
174         s.parseState = s.parseState[0:n]
175         s.redo = false
176         if n == 0 {
177             s.step = stateEndTop
178             s.endTop = true
179         } else {
180             s.step = stateEndValue
181         }
182     }
183
184     func isSpace(c rune) bool {
185         return c == ' ' || c == '\t' || c == '\r' || c == '\
186     }
187
188     // stateBeginValueOrEmpty is the state after reading `[`.
189     func stateBeginValueOrEmpty(s *scanner, c int) int {

```

```

190         if c <= ' ' && isSpace(rune(c)) {
191             return scanSkipSpace
192         }
193         if c == ']' {
194             return stateEndValue(s, c)
195         }
196         return stateBeginValue(s, c)
197     }
198
199     // stateBeginValue is the state at the beginning of the input
200     func stateBeginValue(s *scanner, c int) int {
201         if c <= ' ' && isSpace(rune(c)) {
202             return scanSkipSpace
203         }
204         switch c {
205             case '{':
206                 s.step = stateBeginStringOrEmpty
207                 s.pushParseState(parseObjectKey)
208                 return scanBeginObject
209             case '[':
210                 s.step = stateBeginValueOrEmpty
211                 s.pushParseState(parseArrayValue)
212                 return scanBeginArray
213             case '"':
214                 s.step = stateInString
215                 return scanBeginLiteral
216             case '-':
217                 s.step = stateNeg
218                 return scanBeginLiteral
219             case '0': // beginning of 0.123
220                 s.step = state0
221                 return scanBeginLiteral
222             case 't': // beginning of true
223                 s.step = stateT
224                 return scanBeginLiteral
225             case 'f': // beginning of false
226                 s.step = stateF
227                 return scanBeginLiteral
228             case 'n': // beginning of null
229                 s.step = stateN
230                 return scanBeginLiteral
231         }
232         if '1' <= c && c <= '9' { // beginning of 1234.5
233             s.step = state1
234             return scanBeginLiteral
235         }
236         return s.error(c, "looking for beginning of value")
237     }
238
239     // stateBeginStringOrEmpty is the state after reading `{`.

```

```

240 func stateBeginStringOrEmpty(s *scanner, c int) int {
241     if c <= ' ' && isSpace(rune(c)) {
242         return scanSkipSpace
243     }
244     if c == '}' {
245         n := len(s.parseState)
246         s.parseState[n-1] = parseObjectValue
247         return stateEndValue(s, c)
248     }
249     return stateBeginString(s, c)
250 }
251
252 // stateBeginString is the state after reading `{"key": valu
253 func stateBeginString(s *scanner, c int) int {
254     if c <= ' ' && isSpace(rune(c)) {
255         return scanSkipSpace
256     }
257     if c == '"' {
258         s.step = stateInString
259         return scanBeginLiteral
260     }
261     return s.error(c, "looking for beginning of object k
262 }
263
264 // stateEndValue is the state after completing a value,
265 // such as after reading `{}` or `true` or `["x"`.
266 func stateEndValue(s *scanner, c int) int {
267     n := len(s.parseState)
268     if n == 0 {
269         // Completed top-level before the current by
270         s.step = stateEndTop
271         s.endTop = true
272         return stateEndTop(s, c)
273     }
274     if c <= ' ' && isSpace(rune(c)) {
275         s.step = stateEndValue
276         return scanSkipSpace
277     }
278     ps := s.parseState[n-1]
279     switch ps {
280     case parseObjectKey:
281         if c == ':' {
282             s.parseState[n-1] = parseObjectValue
283             s.step = stateBeginValue
284             return scanObjectKey
285         }
286         return s.error(c, "after object key")
287     case parseObjectValue:
288         if c == ',' {

```

```

289             s.parseState[n-1] = parseObjectKey
290             s.step = stateBeginInitString
291             return scanObjectValue
292         }
293         if c == '}' {
294             s.popParseState()
295             return scanEndObject
296         }
297         return s.error(c, "after object key:value pa
298 case parseArrayValue:
299     if c == ',' {
300         s.step = stateBeginInitValue
301         return scanArrayValue
302     }
303     if c == ']' {
304         s.popParseState()
305         return scanEndArray
306     }
307     return s.error(c, "after array element")
308 }
309 return s.error(c, "")
310 }
311
312 // stateEndTop is the state after finishing the top-level va
313 // such as after reading `{}` or `[1,2,3]`.
314 // Only space characters should be seen now.
315 func stateEndTop(s *scanner, c int) int {
316     if c != ' ' && c != '\t' && c != '\r' && c != '\n' {
317         // Complain about non-space byte on next cal
318         s.error(c, "after top-level value")
319     }
320     return scanEnd
321 }
322
323 // stateInString is the state after reading `""`.
324 func stateInString(s *scanner, c int) int {
325     if c == '"' {
326         s.step = stateEndValue
327         return scanContinue
328     }
329     if c == '\\' {
330         s.step = stateInStringEsc
331         return scanContinue
332     }
333     if c < 0x20 {
334         return s.error(c, "in string literal")
335     }
336     return scanContinue
337 }

```

```

338
339 // stateInStringEsc is the state after reading `"\` during a
340 func stateInStringEsc(s *scanner, c int) int {
341     switch c {
342     case 'b', 'f', 'n', 'r', 't', '\\', '/', "'":
343         s.step = stateInString
344         return scanContinue
345     }
346     if c == 'u' {
347         s.step = stateInStringEscU
348         return scanContinue
349     }
350     return s.error(c, "in string escape code")
351 }
352
353 // stateInStringEscU is the state after reading `"\u` during
354 func stateInStringEscU(s *scanner, c int) int {
355     if '0' <= c && c <= '9' || 'a' <= c && c <= 'f' || '
356         s.step = stateInStringEscU1
357         return scanContinue
358     }
359     // numbers
360     return s.error(c, "in \\u hexadecimal character esca
361 }
362
363 // stateInStringEscU1 is the state after reading `"\u1` duri
364 func stateInStringEscU1(s *scanner, c int) int {
365     if '0' <= c && c <= '9' || 'a' <= c && c <= 'f' || '
366         s.step = stateInStringEscU12
367         return scanContinue
368     }
369     // numbers
370     return s.error(c, "in \\u hexadecimal character esca
371 }
372
373 // stateInStringEscU12 is the state after reading `"\u12` du
374 func stateInStringEscU12(s *scanner, c int) int {
375     if '0' <= c && c <= '9' || 'a' <= c && c <= 'f' || '
376         s.step = stateInStringEscU123
377         return scanContinue
378     }
379     // numbers
380     return s.error(c, "in \\u hexadecimal character esca
381 }
382
383 // stateInStringEscU123 is the state after reading `"\u123`
384 func stateInStringEscU123(s *scanner, c int) int {
385     if '0' <= c && c <= '9' || 'a' <= c && c <= 'f' || '
386         s.step = stateInString
387         return scanContinue

```

```

388     }
389     // numbers
390     return s.error(c, "in \\u hexadecimal character esca
391 }
392
393 // stateInStringEscU123 is the state after reading `-'` durin
394 func stateNeg(s *scanner, c int) int {
395     if c == '0' {
396         s.step = state0
397         return scanContinue
398     }
399     if '1' <= c && c <= '9' {
400         s.step = state1
401         return scanContinue
402     }
403     return s.error(c, "in numeric literal")
404 }
405
406 // state1 is the state after reading a non-zero integer duri
407 // such as after reading `1` or `100` but not `0`.
408 func state1(s *scanner, c int) int {
409     if '0' <= c && c <= '9' {
410         s.step = state1
411         return scanContinue
412     }
413     return state0(s, c)
414 }
415
416 // state0 is the state after reading `0` during a number.
417 func state0(s *scanner, c int) int {
418     if c == '.' {
419         s.step = stateDot
420         return scanContinue
421     }
422     if c == 'e' || c == 'E' {
423         s.step = stateE
424         return scanContinue
425     }
426     return stateEndValue(s, c)
427 }
428
429 // stateDot is the state after reading the integer and decim
430 // such as after reading `1.`.
431 func stateDot(s *scanner, c int) int {
432     if '0' <= c && c <= '9' {
433         s.step = stateDot0
434         return scanContinue
435     }
436     return s.error(c, "after decimal point in numeric li

```

```

437 }
438
439 // stateDot0 is the state after reading the integer, decimal
440 // digits of a number, such as after reading `3.14`.
441 func stateDot0(s *scanner, c int) int {
442     if '0' <= c && c <= '9' {
443         s.step = stateDot0
444         return scanContinue
445     }
446     if c == 'e' || c == 'E' {
447         s.step = stateE
448         return scanContinue
449     }
450     return stateEndValue(s, c)
451 }
452
453 // stateE is the state after reading the mantissa and e in a
454 // such as after reading `314e` or `0.314e`.
455 func stateE(s *scanner, c int) int {
456     if c == '+' {
457         s.step = stateESign
458         return scanContinue
459     }
460     if c == '-' {
461         s.step = stateESign
462         return scanContinue
463     }
464     return stateESign(s, c)
465 }
466
467 // stateESign is the state after reading the mantissa, e, an
468 // such as after reading `314e-` or `0.314e+`.
469 func stateESign(s *scanner, c int) int {
470     if '0' <= c && c <= '9' {
471         s.step = stateE0
472         return scanContinue
473     }
474     return s.error(c, "in exponent of numeric literal")
475 }
476
477 // stateE0 is the state after reading the mantissa, e, optio
478 // and at least one digit of the exponent in a number,
479 // such as after reading `314e-2` or `0.314e+1` or `3.14e0`.
480 func stateE0(s *scanner, c int) int {
481     if '0' <= c && c <= '9' {
482         s.step = stateE0
483         return scanContinue
484     }
485     return stateEndValue(s, c)

```

```

486 }
487
488 // stateT is the state after reading `t`.
489 func stateT(s *scanner, c int) int {
490     if c == 'r' {
491         s.step = stateTr
492         return scanContinue
493     }
494     return s.error(c, "in literal true (expecting 'r')")
495 }
496
497 // stateTr is the state after reading `tr`.
498 func stateTr(s *scanner, c int) int {
499     if c == 'u' {
500         s.step = stateTru
501         return scanContinue
502     }
503     return s.error(c, "in literal true (expecting 'u')")
504 }
505
506 // stateTru is the state after reading `tru`.
507 func stateTru(s *scanner, c int) int {
508     if c == 'e' {
509         s.step = stateEndValue
510         return scanContinue
511     }
512     return s.error(c, "in literal true (expecting 'e')")
513 }
514
515 // stateF is the state after reading `f`.
516 func stateF(s *scanner, c int) int {
517     if c == 'a' {
518         s.step = stateFa
519         return scanContinue
520     }
521     return s.error(c, "in literal false (expecting 'a')")
522 }
523
524 // stateFa is the state after reading `fa`.
525 func stateFa(s *scanner, c int) int {
526     if c == 'l' {
527         s.step = stateFal
528         return scanContinue
529     }
530     return s.error(c, "in literal false (expecting 'l')")
531 }
532
533 // stateFal is the state after reading `fal`.
534 func stateFal(s *scanner, c int) int {
535     if c == 's' {

```

```

536         s.step = stateFals
537         return scanContinue
538     }
539     return s.error(c, "in literal false (expecting 's')")
540 }
541
542 // stateFals is the state after reading `fals`.
543 func stateFals(s *scanner, c int) int {
544     if c == 'e' {
545         s.step = stateEndValue
546         return scanContinue
547     }
548     return s.error(c, "in literal false (expecting 'e')")
549 }
550
551 // stateN is the state after reading `n`.
552 func stateN(s *scanner, c int) int {
553     if c == 'u' {
554         s.step = stateNu
555         return scanContinue
556     }
557     return s.error(c, "in literal null (expecting 'u')")
558 }
559
560 // stateNu is the state after reading `nu`.
561 func stateNu(s *scanner, c int) int {
562     if c == 'l' {
563         s.step = stateNul
564         return scanContinue
565     }
566     return s.error(c, "in literal null (expecting 'l')")
567 }
568
569 // stateNul is the state after reading `nul`.
570 func stateNul(s *scanner, c int) int {
571     if c == 'l' {
572         s.step = stateEndValue
573         return scanContinue
574     }
575     return s.error(c, "in literal null (expecting 'l')")
576 }
577
578 // stateError is the state after reaching a syntax error,
579 // such as after reading `[1]` or `5.1.2`.
580 func stateError(s *scanner, c int) int {
581     return scanError
582 }
583
584 // error records an error and switches to the error state.

```

```

585 func (s *scanner) error(c int, context string) int {
586     s.step = stateError
587     s.err = &SyntaxError{"invalid character " + quoteCha
588     return scanError
589 }
590
591 // quoteChar formats c as a quoted character literal
592 func quoteChar(c int) string {
593     // special cases - different from quoted strings
594     if c == '\\' {
595         return `\\`
596     }
597     if c == '"' {
598         return `"`
599     }
600
601     // use quoted string with different quotation marks
602     s := strconv.Quote(string(c))
603     return "\"" + s[1:len(s)-1] + "\""
604 }
605
606 // undo causes the scanner to return scanCode from the next
607 // This gives callers a simple 1-byte undo mechanism.
608 func (s *scanner) undo(scanCode int) {
609     if s.redo {
610         panic("json: invalid use of scanner")
611     }
612     s.redoCode = scanCode
613     s.redoState = s.step
614     s.step = stateRedo
615     s.redo = true
616 }
617
618 // stateRedo helps implement the scanner's 1-byte undo.
619 func stateRedo(s *scanner, c int) int {
620     s.redo = false
621     s.step = s.redoState
622     return s.redoCode
623 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/json/stream.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package json
6
7 import (
8     "errors"
9     "io"
10 )
11
12 // A Decoder reads and decodes JSON objects from an input st
13 type Decoder struct {
14     r      io.Reader
15     buf    []byte
16     d      decodeState
17     scan  scanner
18     err    error
19 }
20
21 // NewDecoder returns a new decoder that reads from r.
22 //
23 // The decoder introduces its own buffering and may
24 // read data from r beyond the JSON values requested.
25 func NewDecoder(r io.Reader) *Decoder {
26     return &Decoder{r: r}
27 }
28
29 // Decode reads the next JSON-encoded value from its
30 // input and stores it in the value pointed to by v.
31 //
32 // See the documentation for Unmarshal for details about
33 // the conversion of JSON into a Go value.
34 func (dec *Decoder) Decode(v interface{}) error {
35     if dec.err != nil {
36         return dec.err
37     }
38
39     n, err := dec.readValue()
40     if err != nil {
41         return err
```

```

42     }
43
44     // Don't save err from unmarshal into dec.err:
45     // the connection is still usable since we read a co
46     // object from it before the error happened.
47     dec.d.init(dec.buf[0:n])
48     err = dec.d.unmarshal(v)
49
50     // Slide rest of data down.
51     rest := copy(dec.buf, dec.buf[n:])
52     dec.buf = dec.buf[0:rest]
53
54     return err
55 }
56
57 // readValue reads a JSON value into dec.buf.
58 // It returns the length of the encoding.
59 func (dec *Decoder) readValue() (int, error) {
60     dec.scan.reset()
61
62     scanp := 0
63     var err error
64     Input:
65     for {
66         // Look in the buffer for a new value.
67         for i, c := range dec.buf[scanp:] {
68             dec.scan.bytes++
69             v := dec.scan.step(&dec.scan, int(c))
70             if v == scanEnd {
71                 scanp += i
72                 break Input
73             }
74             // scanEnd is delayed one byte.
75             // We might block trying to get that
76             // so instead invent a space byte.
77             if v == scanEndObject && dec.scan.st
78                 scanp += i + 1
79                 break Input
80             }
81             if v == scanError {
82                 dec.err = dec.scan.err
83                 return 0, dec.scan.err
84             }
85         }
86         scanp = len(dec.buf)
87
88         // Did the last read have an error?
89         // Delayed until now to allow buffer scan.
90         if err != nil {
91             if err == io.EOF {

```

```

92             if dec.scan.step(&dec.scan,
93                 break Input
94             }
95             if nonSpace(dec.buf) {
96                 err = io.ErrUnexpected
97             }
98         }
99         dec.err = err
100        return 0, err
101    }
102
103    // Make room to read more into the buffer.
104    const minRead = 512
105    if cap(dec.buf)-len(dec.buf) < minRead {
106        newBuf := make([]byte, len(dec.buf),
107            copy(newBuf, dec.buf)
108            dec.buf = newBuf
109    }
110
111    // Read. Delay error for next iteration (af
112    var n int
113    n, err = dec.r.Read(dec.buf[len(dec.buf):cap
114    dec.buf = dec.buf[0 : len(dec.buf)+n]
115    }
116    return scanp, nil
117 }
118
119 func nonSpace(b []byte) bool {
120     for _, c := range b {
121         if !isSpace(rune(c)) {
122             return true
123         }
124     }
125     return false
126 }
127
128 // An Encoder writes JSON objects to an output stream.
129 type Encoder struct {
130     w io.Writer
131     e encodeState
132     err error
133 }
134
135 // NewEncoder returns a new encoder that writes to w.
136 func NewEncoder(w io.Writer) *Encoder {
137     return &Encoder{w: w}
138 }
139
140 // Encode writes the JSON encoding of v to the connection.

```

```

141 //
142 // See the documentation for Marshal for details about the
143 // conversion of Go values to JSON.
144 func (enc *Encoder) Encode(v interface{}) error {
145     if enc.err != nil {
146         return enc.err
147     }
148     enc.e.Reset()
149     err := enc.e.marshal(v)
150     if err != nil {
151         return err
152     }
153
154     // Terminate each value with a newline.
155     // This makes the output look a little nicer
156     // when debugging, and some kind of space
157     // is required if the encoded value was a number,
158     // so that the reader knows there aren't more
159     // digits coming.
160     enc.e.WriteByte('\n')
161
162     if _, err = enc.w.Write(enc.e.Bytes()); err != nil {
163         enc.err = err
164     }
165     return err
166 }
167
168 // RawMessage is a raw encoded JSON object.
169 // It implements Marshaler and Unmarshaler and can
170 // be used to delay JSON decoding or precompute a JSON encod
171 type RawMessage []byte
172
173 // MarshalJSON returns *m as the JSON encoding of m.
174 func (m *RawMessage) MarshalJSON() ([]byte, error) {
175     return *m, nil
176 }
177
178 // UnmarshalJSON sets *m to a copy of data.
179 func (m *RawMessage) UnmarshalJSON(data []byte) error {
180     if m == nil {
181         return errors.New("json.RawMessage: Unmarsh
182     }
183     *m = append((*m)[0:0], data...)
184     return nil
185 }
186
187 var _ Marshaler = (*RawMessage)(nil)
188 var _ Unmarshaler = (*RawMessage)(nil)

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/json/tags.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package json
6
7 import (
8     "strings"
9 )
10
11 // tagOptions is the string following a comma in a struct fi
12 // tag, or the empty string. It does not include the leading
13 type tagOptions string
14
15 // parseTag splits a struct field's json tag into its name a
16 // comma-separated options.
17 func parseTag(tag string) (string, tagOptions) {
18     if idx := strings.Index(tag, ","); idx != -1 {
19         return tag[:idx], tagOptions(tag[idx+1:])
20     }
21     return tag, tagOptions("")
22 }
23
24 // Contains returns whether checks that a comma-separated li
25 // contains a particular substr flag. substr must be surroun
26 // string boundary or commas.
27 func (o tagOptions) Contains(optionName string) bool {
28     if len(o) == 0 {
29         return false
30     }
31     s := string(o)
32     for s != "" {
33         var next string
34         i := strings.Index(s, ",")
35         if i >= 0 {
36             s, next = s[:i], s[i+1:]
37         }
38         if s == optionName {
39             return true
40         }
41         s = next

```

```
42         }  
43         return false  
44     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/pem/pem.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package pem implements the PEM data encoding, which origi
6 // Enhanced Mail. The most common use of PEM encoding today
7 // certificates. See RFC 1421.
8 package pem
9
10 import (
11     "bytes"
12     "encoding/base64"
13     "io"
14 )
15
16 // A Block represents a PEM encoded structure.
17 //
18 // The encoded form is:
19 //     -----BEGIN Type-----
20 //     Headers
21 //     base64-encoded Bytes
22 //     -----END Type-----
23 // where Headers is a possibly empty sequence of Key: Value
24 type Block struct {
25     Type      string // The type, taken from th
26     Headers  map[string]string // Optional headers.
27     Bytes    []byte // The decoded bytes of th
28 }
29
30 // getLine results the first \r\n or \n delineated line from
31 // array. The line does not include trailing whitespace or t
32 // line bytes. The remainder of the byte array (also not inc
33 // bytes) is also returned and this will always be smaller t
34 // argument.
35 func getLine(data []byte) (line, rest []byte) {
36     i := bytes.Index(data, []byte{'\n'})
37     var j int
38     if i < 0 {
39         i = len(data)
40         j = i
41     } else {
```

```

42         j = i + 1
43         if i > 0 && data[i-1] == '\r' {
44             i--
45         }
46     }
47     return bytes.TrimRight(data[0:i], " \t"), data[j:]
48 }
49
50 // removeWhitespace returns a copy of its input with all spa
51 // newline characters removed.
52 func removeWhitespace(data []byte) []byte {
53     result := make([]byte, len(data))
54     n := 0
55
56     for _, b := range data {
57         if b == ' ' || b == '\t' || b == '\r' || b =
58             continue
59     }
60     result[n] = b
61     n++
62 }
63
64     return result[0:n]
65 }
66
67 var pemStart = []byte("\n-----BEGIN ")
68 var pemEnd = []byte("\n-----END ")
69 var pemEndOfLine = []byte("-----")
70
71 // Decode will find the next PEM formatted block (certificat
72 // etc) in the input. It returns that block and the remainde
73 // no PEM data is found, p is nil and the whole of the input
74 // rest.
75 func Decode(data []byte) (p *Block, rest []byte) {
76     // pemStart begins with a newline. However, at the v
77     // the byte array, we'll accept the start string wit
78     rest = data
79     if bytes.HasPrefix(data, pemStart[1:]) {
80         rest = rest[len(pemStart)-1 : len(data)]
81     } else if i := bytes.Index(data, pemStart); i >= 0 {
82         rest = rest[i+len(pemStart) : len(data)]
83     } else {
84         return nil, data
85     }
86
87     typeLine, rest := getLine(rest)
88     if !bytes.HasSuffix(typeLine, pemEndOfLine) {
89         return decodeError(data, rest)
90     }
91     typeLine = typeLine[0 : len(typeLine)-len(pemEndOfLi

```

```

92
93     p = &Block{
94         Headers: make(map[string]string),
95         Type:    string(typeLine),
96     }
97
98     for {
99         // This loop terminates because getLine's se
100        // always smaller than its argument.
101        if len(rest) == 0 {
102            return nil, data
103        }
104        line, next := getLine(rest)
105
106        i := bytes.Index(line, []byte{':'})
107        if i == -1 {
108            break
109        }
110
111        // TODO(agl): need to cope with values that
112        key, val := line[0:i], line[i+1:]
113        key = bytes.TrimSpace(key)
114        val = bytes.TrimSpace(val)
115        p.Headers[string(key)] = string(val)
116        rest = next
117    }
118
119    i := bytes.Index(rest, pemEnd)
120    if i < 0 {
121        return decodeError(data, rest)
122    }
123    base64Data := removeWhitespace(rest[0:i])
124
125    p.Bytes = make([]byte, base64.StdEncoding.DecodedLen
126    n, err := base64.StdEncoding.Decode(p.Bytes, base64D
127    if err != nil {
128        return decodeError(data, rest)
129    }
130    p.Bytes = p.Bytes[0:n]
131
132    _, rest = getLine(rest[i+len(pemEnd):])
133
134    return
135 }
136
137 func decodeError(data, rest []byte) (*Block, []byte) {
138     // If we get here then we have rejected a likely loc
139     // ultimately invalid PEM block. We need to start ov
140     // position. We have consumed the preamble line and

```

```

141 // any lines which could be header lines. However, a
142 // line is not a valid header line, therefore we can
143 // the preamble line for the any subsequent block. T
144 // find any valid block, no matter what bytes preced
145 //
146 // For example, if the input is
147 //
148 // -----BEGIN MALFORMED BLOCK-----
149 // junk that may look like header lines
150 // or data lines, but no END line
151 //
152 // -----BEGIN ACTUAL BLOCK-----
153 // reldata
154 // -----END ACTUAL BLOCK-----
155 //
156 // we've failed to parse using the first BEGIN line
157 // and now will try again, using the second BEGIN li
158 p, rest := Decode(rest)
159 if p == nil {
160     rest = data
161 }
162 return p, rest
163 }
164
165 const pemLineLength = 64
166
167 type lineBreaker struct {
168     line [pemLineLength]byte
169     used int
170     out io.Writer
171 }
172
173 func (l *lineBreaker) Write(b []byte) (n int, err error) {
174     if l.used+len(b) < pemLineLength {
175         copy(l.line[l.used:], b)
176         l.used += len(b)
177         return len(b), nil
178     }
179
180     n, err = l.out.Write(l.line[0:l.used])
181     if err != nil {
182         return
183     }
184     excess := pemLineLength - l.used
185     l.used = 0
186
187     n, err = l.out.Write(b[0:excess])
188     if err != nil {
189         return

```

```

190     }
191
192     n, err = l.out.Write([]byte{'\n'})
193     if err != nil {
194         return
195     }
196
197     return l.Write(b[excess:])
198 }
199
200 func (l *lineBreaker) Close() (err error) {
201     if l.used > 0 {
202         _, err = l.out.Write(l.line[0:l.used])
203         if err != nil {
204             return
205         }
206         _, err = l.out.Write([]byte{'\n'})
207     }
208
209     return
210 }
211
212 func Encode(out io.Writer, b *Block) (err error) {
213     _, err = out.Write(pemStart[1:])
214     if err != nil {
215         return
216     }
217     _, err = out.Write([]byte(b.Type + "-----\n"))
218     if err != nil {
219         return
220     }
221
222     if len(b.Headers) > 0 {
223         for k, v := range b.Headers {
224             _, err = out.Write([]byte(k + ": " +
225                 v + "\n"))
226             if err != nil {
227                 return
228             }
229         }
230         _, err = out.Write([]byte{'\n'})
231         if err != nil {
232             return
233         }
234
235         var breaker lineBreaker
236         breaker.out = out
237
238         b64 := base64.NewEncoder(base64.StdEncoding, &breaker)
239         _, err = b64.Write(b.Bytes)

```

```
240         if err != nil {
241             return
242         }
243         b64.Close()
244         breaker.Close()
245
246         _, err = out.Write(pemEnd[1:])
247         if err != nil {
248             return
249         }
250         _, err = out.Write([]byte(b.Type + "-----\n"))
251         return
252     }
253
254     func EncodeToMemory(b *Block) []byte {
255         var buf bytes.Buffer
256         Encode(&buf, b)
257         return buf.Bytes()
258     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/xml/marshal.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package xml
6
7 import (
8     "bufio"
9     "bytes"
10    "fmt"
11    "io"
12    "reflect"
13    "strconv"
14    "strings"
15    "time"
16 )
17
18 const (
19     // A generic XML header suitable for use with the ou
20     // This is not automatically added to any output of
21     // it is provided as a convenience.
22     Header = `<?xml version="1.0" encoding="UTF-8"?>` +
23 )
24
25 // Marshal returns the XML encoding of v.
26 //
27 // Marshal handles an array or slice by marshalling each of
28 // Marshal handles a pointer by marshalling the value it poi
29 // pointer is nil, by writing nothing. Marshal handles an i
30 // marshalling the value it contains or, if the interface va
31 // writing nothing. Marshal handles all other data by writi
32 // elements containing the data.
33 //
34 // The name for the XML elements is taken from, in order of
35 // - the tag on the XMLName field, if the data is a stru
36 // - the value of the XMLName field of type xml.Name
37 // - the tag of the struct field used to obtain the data
38 // - the name of the struct field used to obtain the dat
39 // - the name of the marshalled type
40 //
41 // The XML element for a struct contains marshalled elements
```

```

42 // exported fields of the struct, with these exceptions:
43 //   - the XMLName field, described above, is omitted.
44 //   - a field with tag "-" is omitted.
45 //   - a field with tag "name,attr" becomes an attribute with
46 //     the given name in the XML element.
47 //   - a field with tag ",attr" becomes an attribute with
48 //     field name in the in the XML element.
49 //   - a field with tag ",chardata" is written as character data
50 //     not as an XML element.
51 //   - a field with tag ",innerxml" is written verbatim, not
52 //     to the usual marshalling procedure.
53 //   - a field with tag ",comment" is written as an XML comment,
54 //     subject to the usual marshalling procedure. It must contain
55 //     the "--" string within it.
56 //   - a field with a tag including the "omitempty" option
57 //     if the field value is empty. The empty values are a
58 //     nil pointer or interface value, and any array, slice, or
59 //     string of length zero.
60 //   - a non-pointer anonymous struct field is handled as if
61 //     fields of its value were part of the outer struct.
62 //
63 // If a field uses a tag "a>b>c", then the element c will be
64 // parent elements a and b. Fields that appear next to each
65 // the same parent will be enclosed in one XML element.
66 //
67 // See MarshalIndent for an example.
68 //
69 // Marshal will return an error if asked to marshal a channel.
70 func Marshal(v interface{}) ([]byte, error) {
71     var b bytes.Buffer
72     if err := NewEncoder(&b).Encode(v); err != nil {
73         return nil, err
74     }
75     return b.Bytes(), nil
76 }
77
78 // MarshalIndent works like Marshal, but each XML element begins on
79 // an indented line that starts with prefix and is followed by
80 // copies of indent according to the nesting depth.
81 func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error) {
82     var b bytes.Buffer
83     enc := NewEncoder(&b)
84     enc.prefix = prefix
85     enc.indent = indent
86     err := enc.marshalValue(reflect.ValueOf(v), nil)
87     enc.Flush()
88     if err != nil {
89         return nil, err
90     }
91     return b.Bytes(), nil

```

```

92 }
93
94 // An Encoder writes XML data to an output stream.
95 type Encoder struct {
96     printer
97 }
98
99 // NewEncoder returns a new encoder that writes to w.
100 func NewEncoder(w io.Writer) *Encoder {
101     return &Encoder{printer{Writer: bufio.NewWriter(w)}}
102 }
103
104 // Encode writes the XML encoding of v to the stream.
105 //
106 // See the documentation for Marshal for details about the c
107 // of Go values to XML.
108 func (enc *Encoder) Encode(v interface{}) error {
109     err := enc.marshalValue(reflect.ValueOf(v), nil)
110     enc.Flush()
111     return err
112 }
113
114 type printer struct {
115     *bufio.Writer
116     indent    string
117     prefix    string
118     depth    int
119     indentedIn bool
120 }
121
122 // marshalValue writes one or more XML elements representing
123 // If val was obtained from a struct field, finfo must have
124 func (p *printer) marshalValue(val reflect.Value, finfo *fie
125     if !val.IsValid() {
126         return nil
127     }
128     if finfo != nil && finfo.flags&fOmitEmpty != 0 && is
129         return nil
130     }
131
132     kind := val.Kind()
133     typ := val.Type()
134
135     // Drill into pointers/interfaces
136     if kind == reflect.Ptr || kind == reflect.Interface
137         if val.IsNil() {
138             return nil
139         }
140     return p.marshalValue(val.Elem(), finfo)

```

```

141     }
142
143     // Slices and arrays iterate over the elements. They
144     if (kind == reflect.Slice || kind == reflect.Array)
145         for i, n := 0, val.Len(); i < n; i++ {
146             if err := p.marshalValue(val.Index(i)); err != nil {
147                 return err
148             }
149         }
150     return nil
151 }
152
153 tinfo, err := getTypeInfo(typ)
154 if err != nil {
155     return err
156 }
157
158 // Precedence for the XML element name is:
159 // 1. XMLName field in underlying struct;
160 // 2. field name/tag in the struct field; and
161 // 3. type name
162 var xmlns, name string
163 if tinfo.xmlname != nil {
164     xmlname := tinfo.xmlname
165     if xmlname.name != "" {
166         xmlns, name = xmlname.xmlns, xmlname.name
167     } else if v, ok := val.FieldByIndex(xmlname.index); ok {
168         xmlns, name = v.Space, v.LocalName
169     }
170 }
171 if name == "" && finfo != nil {
172     xmlns, name = finfo.xmlns, finfo.name
173 }
174 if name == "" {
175     name = typ.Name()
176     if name == "" {
177         return &UnsupportedTypeError{typ}
178     }
179 }
180
181 p.writeIndent(1)
182 p.WriteByte('<')
183 p.WriteString(name)
184
185 if xmlns != "" {
186     p.WriteString(` xmlns="`)
187     // TODO: EscapeString, to avoid the allocation
188     Escape(p, []byte(xmlns))
189     p.WriteByte('"')

```

```

190     }
191
192     // Attributes
193     for i := range tinfo.fields {
194         finfo := &tinfo.fields[i]
195         if finfo.flags&fAttr == 0 {
196             continue
197         }
198         fv := val.FieldByIndex(finfo.idx)
199         if finfo.flags&fOmitEmpty != 0 && isEmptyVal
200             continue
201     }
202     p.WriteByte(' ')
203     p.WriteString(finfo.name)
204     p.WriteString(`="`)
205     if err := p.marshalSimple(fv.Type(), fv); er
206         return err
207     }
208     p.WriteByte('"')
209 }
210 p.WriteByte('>')
211
212 if val.Kind() == reflect.Struct {
213     err = p.marshalStruct(tinfo, val)
214 } else {
215     err = p.marshalSimple(typ, val)
216 }
217 if err != nil {
218     return err
219 }
220
221 p.writeIndent(-1)
222 p.WriteByte('<')
223 p.WriteByte('/')
224 p.WriteString(name)
225 p.WriteByte('>')
226
227 return nil
228 }
229
230 var timeType = reflect.TypeOf(time.Time{})
231
232 func (p *printer) marshalSimple(typ reflect.Type, val reflect
233 // Normally we don't see structs, but this can happen
234 if val.Type() == timeType {
235     p.WriteString(val.Interface().(time.Time).Fo
236     return nil
237 }
238 switch val.Kind() {
239 case reflect.Int, reflect.Int8, reflect.Int16, refle

```



```

289         }
290         continue
291
292     case fComment:
293         k := vf.Kind()
294         if !(k == reflect.String || k == ref
295             return fmt.Errorf("xml: bad
296         }
297         if vf.Len() == 0 {
298             continue
299         }
300         p.writeIndent(0)
301         p.WriteString("<!--")
302         dashDash := false
303         dashLast := false
304         switch k {
305         case reflect.String:
306             s := vf.String()
307             dashDash = strings.Index(s,
308                 dashLast = s[len(s)-1] == '-'
309             if !dashDash {
310                 p.WriteString(s)
311             }
312         case reflect.Slice:
313             b := vf.Bytes()
314             dashDash = bytes.Index(b, dd
315             dashLast = b[len(b)-1] == '-'
316             if !dashDash {
317                 p.Write(b)
318             }
319         default:
320             panic("can't happen")
321         }
322         if dashDash {
323             return fmt.Errorf(`xml: comm
324         }
325         if dashLast {
326             // "--->" is invalid grammar
327             p.WriteByte(' ')
328         }
329         p.WriteString("-->")
330         continue
331
332     case fInnerXml:
333         iface := vf.Interface()
334         switch raw := iface.(type) {
335         case []byte:
336             p.Write(raw)
337             continue

```

```

338         case string:
339             p.WriteString(raw)
340             continue
341         }
342
343         case fElement:
344             s.trim(finfo.parents)
345             if len(finfo.parents) > len(s.stack)
346                 if vf.Kind() != reflect.Ptr
347                     s.push(finfo.parents)
348             }
349         }
350     }
351     if err := p.marshalValue(vf, finfo); err !=
352         return err
353     }
354 }
355 s.trim(nil)
356 return nil
357 }
358
359 func (p *printer) writeIndent(depthDelta int) {
360     if len(p.prefix) == 0 && len(p.indent) == 0 {
361         return
362     }
363     if depthDelta < 0 {
364         p.depth--
365         if p.indentedIn {
366             p.indentedIn = false
367             return
368         }
369         p.indentedIn = false
370     }
371     p.WriteByte('\n')
372     if len(p.prefix) > 0 {
373         p.WriteString(p.prefix)
374     }
375     if len(p.indent) > 0 {
376         for i := 0; i < p.depth; i++ {
377             p.WriteString(p.indent)
378         }
379     }
380     if depthDelta > 0 {
381         p.depth++
382         p.indentedIn = true
383     }
384 }
385
386 type parentStack struct {
387     *printer

```

```

388         stack []string
389     }
390
391     // trim updates the XML context to match the longest common
392     // and the given parents. A closing tag will be written for
393     // popped. Passing a zero slice or nil will close all the e
394     func (s *parentStack) trim(parents []string) {
395         split := 0
396         for ; split < len(parents) && split < len(s.stack);
397             if parents[split] != s.stack[split] {
398                 break
399             }
400     }
401     for i := len(s.stack) - 1; i >= split; i-- {
402         s.writeIndent(-1)
403         s.WriteString("</")
404         s.WriteString(s.stack[i])
405         s.WriteByte('>')
406     }
407     s.stack = parents[:split]
408 }
409
410 // push adds parent elements to the stack and writes open ta
411 func (s *parentStack) push(parents []string) {
412     for i := 0; i < len(parents); i++ {
413         s.writeIndent(1)
414         s.WriteByte('<')
415         s.WriteString(parents[i])
416         s.WriteByte('>')
417     }
418     s.stack = append(s.stack, parents...)
419 }
420
421 // A MarshalXMLError is returned when Marshal encounters a t
422 // that cannot be converted into XML.
423 type UnsupportedTypeError struct {
424     Type reflect.Type
425 }
426
427 func (e *UnsupportedTypeError) Error() string {
428     return "xml: unsupported type: " + e.Type.String()
429 }
430
431 func isEmptyValue(v reflect.Value) bool {
432     switch v.Kind() {
433     case reflect.Array, reflect.Map, reflect.Slice, refl
434         return v.Len() == 0
435     case reflect.Bool:
436         return !v.Bool()

```

```
437     case reflect.Int, reflect.Int8, reflect.Int16, refle
438         return v.Int() == 0
439     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
440         return v.Uint() == 0
441     case reflect.Float32, reflect.Float64:
442         return v.Float() == 0
443     case reflect.Interface, reflect.Ptr:
444         return v.IsNil()
445     }
446     return false
447 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/xml/read.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package xml
6
7 import (
8     "bytes"
9     "errors"
10    "reflect"
11    "strconv"
12    "strings"
13    "time"
14 )
15
16 // BUG(rsc): Mapping between XML elements and data structure
17 // an XML element is an order-dependent collection of anonym
18 // values, while a data structure is an order-independent co
19 // of named values.
20 // See package json for a textual representation more suitab
21 // to data structures.
22
23 // Unmarshal parses the XML-encoded data and stores the resu
24 // the value pointed to by v, which must be an arbitrary str
25 // slice, or string. Well-formed data that does not fit into
26 // discarded.
27 //
28 // Because Unmarshal uses the reflect package, it can only a
29 // to exported (upper case) fields. Unmarshal uses a case-s
30 // comparison to match XML element names to tag values and s
31 // field names.
32 //
33 // Unmarshal maps an XML element to a struct using the follo
34 // In the rules, the tag of a field refers to the value asso
35 // key 'xml' in the struct field's tag (see the example abov
36 //
37 // * If the struct has a field of type []byte or string wi
38 //     ",innerxml", Unmarshal accumulates the raw XML neste
39 //     element in that field. The rest of the rules still
40 //
41 // * If the struct has a field named XMLName of type xml.N
```

```
42 //      Unmarshal records the element name in that field.
43 //
44 //      * If the XMLName field has an associated tag of the for
45 //      "name" or "namespace-URL name", the XML element must
46 //      the given name (and, optionally, name space) or else
47 //      returns an error.
48 //
49 //      * If the XML element has an attribute whose name matches
50 //      struct field name with an associated tag containing
51 //      the explicit name in a struct field tag of the form
52 //      Unmarshal records the attribute value in that field.
53 //
54 //      * If the XML element contains character data, that data
55 //      accumulated in the first struct field that has tag "
56 //      The struct field may have type []byte or string.
57 //      If there is no such field, the character data is discarded.
58 //
59 //      * If the XML element contains comments, they are accumulated in
60 //      the first struct field that has tag ",comments". The
61 //      field may have type []byte or string. If there is no such
62 //      field, the comments are discarded.
63 //
64 //      * If the XML element contains a sub-element whose name matches
65 //      the prefix of a tag formatted as "a" or "a>b>c", unmarshal
66 //      will descend into the XML structure looking for elements with
67 //      given names, and will map the innermost elements to the struct
68 //      field. A tag starting with ">" is equivalent to one
69 //      with the field name followed by ">".
70 //
71 //      * If the XML element contains a sub-element whose name matches
72 //      a struct field's XMLName tag and the struct field has an
73 //      explicit name tag as per the previous rule, unmarshal maps
74 //      the sub-element to that struct field.
75 //
76 //      * If the XML element contains a sub-element whose name matches a
77 //      field without any mode flags ("attr", "chardata", "comment", "text"),
78 //      maps the sub-element to that struct field.
79 //
80 //      * If the XML element contains a sub-element that hasn't matched
81 //      any of the above rules and the struct has a field with tag "-"
82 //      unmarshal maps the sub-element to that struct field.
83 //
84 //      * A non-pointer anonymous struct field is handled as if it had
85 //      fields of its value were part of the outer struct.
86 //
87 //      * A struct field with tag "-" is never unmarshalled into.
88 //
89 //      Unmarshal maps an XML element to a string or []byte by saving the
90 //      concatenation of that element's character data in the struct field.
91 //      []byte. The saved []byte is never nil.
```

```

92 //
93 // Unmarshal maps an attribute value to a string or []byte b
94 // the value in the string or slice.
95 //
96 // Unmarshal maps an XML element to a slice by extending the
97 // the slice and mapping the element to the newly created va
98 //
99 // Unmarshal maps an XML element or attribute value to a boo
100 // setting it to the boolean value represented by the string
101 //
102 // Unmarshal maps an XML element or attribute value to an in
103 // floating-point field by setting the field to the result o
104 // interpreting the string value in decimal. There is no ch
105 // overflow.
106 //
107 // Unmarshal maps an XML element to an xml.Name by recording
108 // element name.
109 //
110 // Unmarshal maps an XML element to a pointer by setting the
111 // to a freshly allocated value and then mapping the element
112 //
113 func Unmarshal(data []byte, v interface{}) error {
114     return NewDecoder(bytes.NewBuffer(data)).Decode(v)
115 }
116
117 // Decode works like xml.Unmarshal, except it reads the deco
118 // stream to find the start element.
119 func (d *Decoder) Decode(v interface{}) error {
120     return d.DecodeElement(v, nil)
121 }
122
123 // DecodeElement works like xml.Unmarshal except that it tak
124 // a pointer to the start XML element to decode into v.
125 // It is useful when a client reads some raw XML tokens itse
126 // but also wants to defer to Unmarshal for some elements.
127 func (d *Decoder) DecodeElement(v interface{}, start *StartE
128     val := reflect.ValueOf(v)
129     if val.Kind() != reflect.Ptr {
130         return errors.New("non-pointer passed to Unr
131     }
132     return d.unmarshal(val.Elem(), start)
133 }
134
135 // An UnmarshalError represents an error in the unmarshalling
136 type UnmarshalError string
137
138 func (e UnmarshalError) Error() string { return string(e) }
139
140 // Unmarshal a single XML element into val.

```

```

141 func (p *Decoder) unmarshal(val reflect.Value, start *StartE
142 // Find start element if we need it.
143 if start == nil {
144     for {
145         tok, err := p.Token()
146         if err != nil {
147             return err
148         }
149         if t, ok := tok.(StartElement); ok {
150             start = &t
151             break
152         }
153     }
154 }
155
156 if pv := val; pv.Kind() == reflect.Ptr {
157     if pv.IsNil() {
158         pv.Set(reflect.New(pv.Type().Elem()))
159     }
160     val = pv.Elem()
161 }
162
163 var (
164     data          []byte
165     saveData      reflect.Value
166     comment       []byte
167     saveComment   reflect.Value
168     saveXML       reflect.Value
169     saveXMLIndex int
170     saveXMLData   []byte
171     saveAny       reflect.Value
172     sv            reflect.Value
173     tinfo         *TypeInfo
174     err           error
175 )
176
177 switch v := val; v.Kind() {
178 default:
179     return errors.New("unknown type " + v.Type())
180
181 case reflect.Interface:
182     // TODO: For now, simply ignore the field. I
183     //       future we may choose to unmarshal t
184     //       element on it, if not nil.
185     return p.Skip()
186
187 case reflect.Slice:
188     typ := v.Type()
189     if typ.Elem().Kind() == reflect.Uint8 {

```

```

190             // []byte
191             saveData = v
192             break
193         }
194
195         // Slice of element values.
196         // Grow slice.
197         n := v.Len()
198         if n >= v.Cap() {
199             ncap := 2 * n
200             if ncap < 4 {
201                 ncap = 4
202             }
203             new := reflect.MakeSlice(typ, n, ncap)
204             reflect.Copy(new, v)
205             v.Set(new)
206         }
207         v.SetLen(n + 1)
208
209         // Recur to read element into slice.
210         if err := p.unmarshal(v.Index(n), start); err != nil {
211             v.SetLen(n)
212             return err
213         }
214         return nil
215
216     case reflect.Bool, reflect.Float32, reflect.Float64,
217         saveData = v
218
219     case reflect.Struct:
220         typ := v.Type()
221         if typ == nameType {
222             v.Set(reflect.ValueOf(start.Name))
223             break
224         }
225         if typ == timeType {
226             saveData = v
227             break
228         }
229
230         sv = v
231         tinfo, err = getTypeInfo(typ)
232         if err != nil {
233             return err
234         }
235
236         // Validate and assign element name.
237         if tinfo.xmlname != nil {
238             finfo := tinfo.xmlname
239             if finfo.name != "" && finfo.name !=

```

```

240         return UnmarshalError("expec
241     }
242     if finfo.xmlns != "" && finfo.xmlns
243         e := "expected element <" +
244         if start.Name.Space == "" {
245             e += "no name space"
246         } else {
247             e += start.Name.Spac
248         }
249         return UnmarshalError(e)
250     }
251     fv := sv.FieldByIndex(finfo.idx)
252     if _, ok := fv.Interface().(Name); o
253         fv.Set(reflect.ValueOf(start
254     }
255 }
256
257 // Assign attributes.
258 // Also, determine whether we need to save c
259 for i := range tinfo.fields {
260     finfo := &tinfo.fields[i]
261     switch finfo.flags & fMode {
262     case fAttr:
263         strv := sv.FieldByIndex(fin
264         // Look for attribute.
265         for _, a := range start.Attr
266             if a.Name.Local == f
267                 copyValue(st
268                 break
269             }
270         }
271
272     case fCharData:
273         if !saveData.IsValid() {
274             saveData = sv.FieldB
275         }
276
277     case fComment:
278         if !saveComment.IsValid() {
279             saveComment = sv.Fie
280         }
281
282     case fAny:
283         if !saveAny.IsValid() {
284             saveAny = sv.FieldBy
285         }
286
287     case fInnerXml:
288         if !saveXML.IsValid() {

```

```

289             saveXML = sv.FieldBy
290             if p.saved == nil {
291                 saveXMLIndex
292                 p.saved = ne
293             } else {
294                 saveXMLIndex
295             }
296         }
297     }
298 }
299 }
300
301 // Find end element.
302 // Process sub-elements along the way.
303 Loop:
304     for {
305         var savedOffset int
306         if saveXML.IsValid() {
307             savedOffset = p.savedOffset()
308         }
309         tok, err := p.Token()
310         if err != nil {
311             return err
312         }
313         switch t := tok.(type) {
314         case StartElement:
315             consumed := false
316             if sv.IsValid() {
317                 consumed, err = p.unmarshalP
318                 if err != nil {
319                     return err
320                 }
321                 if !consumed && saveAny.IsVa
322                     consumed = true
323                 if err := p.unmarsha
324                     return err
325             }
326         }
327         if !consumed {
328             if err := p.Skip(); err != n
329                 return err
330         }
331     }
332 }
333
334 case EndElement:
335     if saveXML.IsValid() {
336         saveXMLData = p.saved.Bytes(
337         if saveXMLIndex == 0 {

```

```

338                                     p.saved = nil
339                                     }
340                                 }
341                                 break Loop
342
343                             case CharData:
344                                 if saveData.IsValid() {
345                                     data = append(data, t...)
346                                 }
347
348                             case Comment:
349                                 if saveComment.IsValid() {
350                                     comment = append(comment, t.
351                                     )
352                                 }
353                         }
354
355                     if err := copyValue(saveData, data); err != nil {
356                         return err
357                     }
358
359                     switch t := saveComment; t.Kind() {
360                     case reflect.String:
361                         t.SetString(string(comment))
362                     case reflect.Slice:
363                         t.Set(reflect.ValueOf(comment))
364                     }
365
366                     switch t := saveXML; t.Kind() {
367                     case reflect.String:
368                         t.SetString(string(saveXMLData))
369                     case reflect.Slice:
370                         t.Set(reflect.ValueOf(saveXMLData))
371                     }
372
373                     return nil
374                 }
375
376 func copyValue(dst reflect.Value, src []byte) (err error) {
377     // Helper functions for integer and unsigned integer
378     var itmp int64
379     getInt64 := func() bool {
380         itmp, err = strconv.ParseInt(string(src), 10
381         // TODO: should check sizes
382         return err == nil
383     }
384     var utmp uint64
385     getUint64 := func() bool {
386         utmp, err = strconv.ParseUint(string(src), 1
387         // TODO: check for overflow?

```

```

388         return err == nil
389     }
390     var ftmp float64
391     getFloat64 := func() bool {
392         ftmp, err = strconv.ParseFloat(string(src),
393             // TODO: check for overflow?
394             return err == nil
395     }
396
397     // Save accumulated data.
398     switch t := dst; t.Kind() {
399     case reflect.Invalid:
400         // Probably a comment.
401     default:
402         return errors.New("cannot happen: unknown ty
403     case reflect.Int, reflect.Int8, reflect.Int16, refle
404         if !getInt64() {
405             return err
406         }
407         t.SetInt(itmp)
408     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
409         if !getUint64() {
410             return err
411         }
412         t.SetUint(utmp)
413     case reflect.Float32, reflect.Float64:
414         if !getFloat64() {
415             return err
416         }
417         t.SetFloat(ftmp)
418     case reflect.Bool:
419         value, err := strconv.ParseBool(strings.Trim
420         if err != nil {
421             return err
422         }
423         t.SetBool(value)
424     case reflect.String:
425         t.SetString(string(src))
426     case reflect.Slice:
427         if len(src) == 0 {
428             // non-nil to flag presence
429             src = []byte{}
430         }
431         t.SetBytes(src)
432     case reflect.Struct:
433         if t.Type() == timeType {
434             tv, err := time.Parse(time.RFC3339,
435             if err != nil {
436                 return err

```

```

437         }
438         t.Set(reflect.ValueOf(tv))
439     }
440 }
441     return nil
442 }
443
444 // unmarshalPath walks down an XML structure looking for wan
445 // paths, and calls unmarshal on them.
446 // The consumed result tells whether XML elements have been
447 // from the Decoder until start's matching end element, or i
448 // still untouched because start is uninteresting for sv's f
449 func (p *Decoder) unmarshalPath(tinfo *TypeInfo, sv reflect.
450     recurse := false
451 Loop:
452     for i := range tinfo.fields {
453         finfo := &tinfo.fields[i]
454         if finfo.flags&fElement == 0 || len(finfo.pa
455             continue
456         }
457         for j := range parents {
458             if parents[j] != finfo.parents[j] {
459                 continue Loop
460             }
461         }
462         if len(finfo.parents) == len(parents) && fin
463             // It's a perfect match, unmarshal t
464             return true, p.unmarshal(sv.FieldByI
465         }
466         if len(finfo.parents) > len(parents) && finf
467             // It's a prefix for the field. Brea
468             // since it's not ok for one field p
469             // the prefix for another field path
470             recurse = true
471
472             // We can reuse the same slice as lo
473             // don't try to append to it.
474             parents = finfo.parents[:len(parents)
475             break
476         }
477     }
478     if !recurse {
479         // We have no business with this element.
480         return false, nil
481     }
482     // The element is not a perfect match for any field,
483     // or more fields have the path to this element as a
484     // prefix. Recurse and attempt to match these.
485     for {

```

```

486         var tok Token
487         tok, err = p.Token()
488         if err != nil {
489             return true, err
490         }
491         switch t := tok.(type) {
492         case StartElement:
493             consumed2, err := p.unmarshalPath(ti
494             if err != nil {
495                 return true, err
496             }
497             if !consumed2 {
498                 if err := p.Skip(); err != n
499                 return true, err
500             }
501         }
502         case EndElement:
503             return true, nil
504         }
505     }
506     panic("unreachable")
507 }
508
509 // Skip reads tokens until it has consumed the end element
510 // matching the most recent start element already consumed.
511 // It recurs if it encounters a start element, so it can be
512 // skip nested structures.
513 // It returns nil if it finds an end element matching the st
514 // element; otherwise it returns an error describing the pro
515 func (d *Decoder) Skip() error {
516     for {
517         tok, err := d.Token()
518         if err != nil {
519             return err
520         }
521         switch tok.(type) {
522         case StartElement:
523             if err := d.Skip(); err != nil {
524                 return err
525             }
526         case EndElement:
527             return nil
528         }
529     }
530     panic("unreachable")
531 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/xml/typeinfo.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package xml
6
7 import (
8     "fmt"
9     "reflect"
10    "strings"
11    "sync"
12 )
13
14 // typeInfo holds details for the xml representation of a ty
15 type typeInfo struct {
16     xmlname *fieldInfo
17     fields []fieldInfo
18 }
19
20 // fieldInfo holds details for the xml representation of a s
21 type fieldInfo struct {
22     idx []int
23     name string
24     xmlns string
25     flags fieldFlags
26     parents []string
27 }
28
29 type fieldFlags int
30
31 const (
32     fElement fieldFlags = 1 << iota
33     fAttr
34     fCharData
35     fInnerXml
36     fComment
37     fAny
38
39     fOmitEmpty
40
41     fMode = fElement | fAttr | fCharData | fInnerXml | f
```

```

42 )
43
44 var tinfoMap = make(map[reflect.Type]*typeInfo)
45 var tinfoLock sync.RWMutex
46
47 var nameType = reflect.TypeOf(Name{})
48
49 // getTypeInfo returns the typeInfo structure with details n
50 // for marshalling and unmarshalling typ.
51 func getTypeInfo(typ reflect.Type) (*typeInfo, error) {
52     tinfoLock.RLock()
53     tinfo, ok := tinfoMap[typ]
54     tinfoLock.RUnlock()
55     if ok {
56         return tinfo, nil
57     }
58     tinfo = &typeInfo{}
59     if typ.Kind() == reflect.Struct && typ != nameType {
60         n := typ.NumField()
61         for i := 0; i < n; i++ {
62             f := typ.Field(i)
63             if f.PkgPath != "" || f.Tag.Get("xml")
64                 continue // Private field
65         }
66
67         // For embedded structs, embed its f
68         if f.Anonymous {
69             if f.Type.Kind() != reflect.
70                 continue
71         }
72         inner, err := getTypeInfo(f.
73         if err != nil {
74             return nil, err
75         }
76         for _, finfo := range inner.
77             finfo.idx = append([
78             if err := addFieldIn
79                 return nil,
80         }
81     }
82     continue
83 }
84
85 finfo, err := structFieldInfo(typ, &
86 if err != nil {
87     return nil, err
88 }
89
90 if f.Name == "XMLName" {
91     tinfo.xmlname = finfo

```

```

92             continue
93         }
94
95         // Add the field if it doesn't conflict
96         if err := addFieldInfo(typ, tinfo, f); err != nil {
97             return nil, err
98         }
99     }
100 }
101 tinfoLock.Lock()
102 tinfoMap[typ] = tinfo
103 tinfoLock.Unlock()
104 return tinfo, nil
105 }
106
107 // structFieldInfo builds and returns a fieldInfo for f.
108 func structFieldInfo(typ reflect.Type, f *reflect.StructField) fieldInfo {
109     finfo := &fieldInfo{idx: f.Index}
110
111     // Split the tag from the xml namespace if necessary
112     tag := f.Tag.Get("xml")
113     if i := strings.Index(tag, " "); i >= 0 {
114         finfo.xmlns, tag = tag[:i], tag[i+1:]
115     }
116
117     // Parse flags.
118     tokens := strings.Split(tag, ",")
119     if len(tokens) == 1 {
120         finfo.flags = fElement
121     } else {
122         tag = tokens[0]
123         for _, flag := range tokens[1:] {
124             switch flag {
125             case "attr":
126                 finfo.flags |= fAttr
127             case "chardata":
128                 finfo.flags |= fCharData
129             case "innerxml":
130                 finfo.flags |= fInnerXml
131             case "comment":
132                 finfo.flags |= fComment
133             case "any":
134                 finfo.flags |= fAny
135             case "omitempty":
136                 finfo.flags |= fOmitEmpty
137             }
138         }
139
140         // Validate the flags used.

```

```

141         valid := true
142         switch mode := finfo.flags & fMode; mode {
143         case 0:
144             finfo.flags |= fElement
145         case fAttr, fCharData, fInnerXml, fComment,
146             if f.Name == "XMLName" || tag != ""
147                 valid = false
148             }
149         default:
150             // This will also catch multiple mod
151             valid = false
152         }
153         if finfo.flags&fOmitEmpty != 0 && finfo.flag
154             valid = false
155         }
156         if !valid {
157             return nil, fmt.Errorf("xml: invalid
158                 f.Name, typ, f.Tag.Get("xml"
159             )
160         }
161     }
162     // Use of xmlns without a name is not allowed.
163     if finfo.xmlns != "" && tag == "" {
164         return nil, fmt.Errorf("xml: namespace witho
165             f.Name, typ, f.Tag.Get("xml"))
166     }
167
168     if f.Name == "XMLName" {
169         // The XMLName field records the XML element
170         // process it as usual because its name shou
171         // empty rather than to the field name.
172         finfo.name = tag
173         return finfo, nil
174     }
175
176     if tag == "" {
177         // If the name part of the tag is completely
178         // default from XMLName of underlying struct
179         // or field name otherwise.
180         if xmlname := lookupXMLName(f.Type); xmlname
181             finfo.xmlns, finfo.name = xmlname.xml
182         } else {
183             finfo.name = f.Name
184         }
185         return finfo, nil
186     }
187
188     // Prepare field name and parents.
189     tokens = strings.Split(tag, ">")

```

```

190     if tokens[0] == "" {
191         tokens[0] = f.Name
192     }
193     if tokens[len(tokens)-1] == "" {
194         return nil, fmt.Errorf("xml: trailing '>' in
195     }
196     finfo.name = tokens[len(tokens)-1]
197     if len(tokens) > 1 {
198         finfo.parents = tokens[:len(tokens)-1]
199     }
200
201     // If the field type has an XMLName field, the names
202     // so that the behavior of both marshalling and unma
203     // is straightforward and unambiguous.
204     if finfo.flags&fElement != 0 {
205         ftyp := f.Type
206         xmlname := lookupXMLName(ftyp)
207         if xmlname != nil && xmlname.name != finfo.n
208             return nil, fmt.Errorf("xml: name %q
209                 finfo.name, typ, f.Name, xml
210         }
211     }
212     return finfo, nil
213 }
214
215 // lookupXMLName returns the fieldInfo for typ's XMLName fie
216 // in case it exists and has a valid xml field tag, otherwis
217 // it returns nil.
218 func lookupXMLName(typ reflect.Type) (xmlname *fieldInfo) {
219     for typ.Kind() == reflect.Ptr {
220         typ = typ.Elem()
221     }
222     if typ.Kind() != reflect.Struct {
223         return nil
224     }
225     for i, n := 0, typ.NumField(); i < n; i++ {
226         f := typ.Field(i)
227         if f.Name != "XMLName" {
228             continue
229         }
230         finfo, err := structFieldInfo(typ, &f)
231         if finfo.name != "" && err == nil {
232             return finfo
233         }
234         // Also consider errors as a non-existent fi
235         // and let getTypeInfo itself report the err
236         break
237     }
238     return nil
239 }

```

```

240
241 func min(a, b int) int {
242     if a <= b {
243         return a
244     }
245     return b
246 }
247
248 // addFieldInfo adds finfo to tinfo.fields if there are no
249 // conflicts, or if conflicts arise from previous fields tha
250 // obtained from deeper embedded structures than finfo. In t
251 // case, the conflicting entries are dropped.
252 // A conflict occurs when the path (parent + name) to a fiel
253 // itself a prefix of another path, or when two paths match
254 // It is okay for field paths to share a common, shorter pre
255 func addFieldInfo(typ reflect.Type, tinfo *typeInfo, newf *f
256     var conflicts []int
257 Loop:
258     // First, figure all conflicts. Most working code wi
259     for i := range tinfo.fields {
260         oldf := &tinfo.fields[i]
261         if oldf.flags&fMode != newf.flags&fMode {
262             continue
263         }
264         minl := min(len(newf.parents), len(oldf.pare
265         for p := 0; p < minl; p++ {
266             if oldf.parents[p] != newf.parents[p]
267                 continue Loop
268         }
269     }
270     if len(oldf.parents) > len(newf.parents) {
271         if oldf.parents[len(newf.parents)] =
272             conflicts = append(conflicts
273     }
274     } else if len(oldf.parents) < len(newf.paren
275         if newf.parents[len(oldf.parents)] =
276             conflicts = append(conflicts
277     }
278     } else {
279         if newf.name == oldf.name {
280             conflicts = append(conflicts
281         }
282     }
283 }
284 // Without conflicts, add the new field and return.
285 if conflicts == nil {
286     tinfo.fields = append(tinfo.fields, *newf)
287     return nil
288 }

```

```

289
290 // If any conflict is shallower, ignore the new fiel
291 // This matches the Go field resolution on embedding
292 for _, i := range conflicts {
293     if len(tinfo.fields[i].idx) < len(newf.idx)
294         return nil
295     }
296 }
297
298 // Otherwise, if any of them is at the same depth le
299 for _, i := range conflicts {
300     oldf := &tinfo.fields[i]
301     if len(oldf.idx) == len(newf.idx) {
302         f1 := typ.FieldByIndex(oldf.idx)
303         f2 := typ.FieldByIndex(newf.idx)
304         return &TagPathError{typ, f1.Name, f
305     }
306 }
307
308 // Otherwise, the new field is shallower, and thus t
309 // so drop the conflicting fields from tinfo and app
310 for c := len(conflicts) - 1; c >= 0; c-- {
311     i := conflicts[c]
312     copy(tinfo.fields[i:], tinfo.fields[i+1:])
313     tinfo.fields = tinfo.fields[:len(tinfo.field
314 }
315 tinfo.fields = append(tinfo.fields, *newf)
316 return nil
317 }
318
319 // A TagPathError represents an error in the unmarshalling p
320 // caused by the use of field tags with conflicting paths.
321 type TagPathError struct {
322     Struct      reflect.Type
323     Field1, Tag1 string
324     Field2, Tag2 string
325 }
326
327 func (e *TagPathError) Error() string {
328     return fmt.Sprintf("%s field %q with tag %q conflict
329 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/encoding/xml/xml.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package xml implements a simple XML 1.0 parser that
6 // understands XML name spaces.
7 package xml
8
9 // References:
10 //   Annotated XML spec: http://www.xml.com/axml/testaxml.h
11 //   XML name spaces: http://www.w3.org/TR/REC-xml-names/
12
13 // TODO(rsc):
14 //   Test error handling.
15
16 import (
17     "bufio"
18     "bytes"
19     "fmt"
20     "io"
21     "strconv"
22     "strings"
23     "unicode"
24     "unicode/utf8"
25 )
26
27 // A SyntaxError represents a syntax error in the XML input
28 type SyntaxError struct {
29     Msg string
30     Line int
31 }
32
33 func (e *SyntaxError) Error() string {
34     return "XML syntax error on line " + strconv.Itoa(e.
35 }
36
37 // A Name represents an XML name (Local) annotated
38 // with a name space identifier (Space).
39 // In tokens returned by Decoder.Token, the Space identifier
40 // is given as a canonical URL, not the short prefix used
41 // in the document being parsed.
```

```

42 type Name struct {
43     Space, Local string
44 }
45
46 // An Attr represents an attribute in an XML element (Name=v
47 type Attr struct {
48     Name Name
49     Value string
50 }
51
52 // A Token is an interface holding one of the token types:
53 // StartElement, EndElement, CharData, Comment, ProcInst, or
54 type Token interface{}
55
56 // A StartElement represents an XML start element.
57 type StartElement struct {
58     Name Name
59     Attr []Attr
60 }
61
62 func (e StartElement) Copy() StartElement {
63     attrs := make([]Attr, len(e.Attr))
64     copy(attrs, e.Attr)
65     e.Attr = attrs
66     return e
67 }
68
69 // An EndElement represents an XML end element.
70 type EndElement struct {
71     Name Name
72 }
73
74 // A CharData represents XML character data (raw text),
75 // in which XML escape sequences have been replaced by
76 // the characters they represent.
77 type CharData []byte
78
79 func makeCopy(b []byte) []byte {
80     b1 := make([]byte, len(b))
81     copy(b1, b)
82     return b1
83 }
84
85 func (c CharData) Copy() CharData { return CharData(makeCopy
86
87 // A Comment represents an XML comment of the form <!--comme
88 // The bytes do not include the <!-- and --> comment markers
89 type Comment []byte
90
91 func (c Comment) Copy() Comment { return Comment(makeCopy(c)

```

```

92
93 // A ProcInst represents an XML processing instruction of th
94 type ProcInst struct {
95     Target string
96     Inst []byte
97 }
98
99 func (p ProcInst) Copy() ProcInst {
100     p.Inst = makeCopy(p.Inst)
101     return p
102 }
103
104 // A Directive represents an XML directive of the form <!tex
105 // The bytes do not include the <! and > markers.
106 type Directive []byte
107
108 func (d Directive) Copy() Directive { return Directive(makeC
109
110 // CopyToken returns a copy of a Token.
111 func CopyToken(t Token) Token {
112     switch v := t.(type) {
113     case CharData:
114         return v.Copy()
115     case Comment:
116         return v.Copy()
117     case Directive:
118         return v.Copy()
119     case ProcInst:
120         return v.Copy()
121     case StartElement:
122         return v.Copy()
123     }
124     return t
125 }
126
127 // A Decoder represents an XML parser reading a particular i
128 // The parser assumes that its input is encoded in UTF-8.
129 type Decoder struct {
130     // Strict defaults to true, enforcing the requiremen
131     // of the XML specification.
132     // If set to false, the parser allows input containi
133     // mistakes:
134     //     * If an element is missing an end tag, the p
135     //     end tags as necessary to keep the return v
136     //     properly balanced.
137     //     * In attribute values and character data, un
138     //     character entities (sequences beginning wi
139     //
140     // Setting:

```

```

141         //
142         //     d.Strict = false;
143         //     d.AutoClose = HTMLAutoClose;
144         //     d.Entity = HTMLEntity
145         //
146         // creates a parser that can handle typical HTML.
147         Strict bool
148
149         // When Strict == false, AutoClose indicates a set o
150         // consider closed immediately after they are opened
151         // of whether an end element is present.
152         AutoClose []string
153
154         // Entity can be used to map non-standard entity nam
155         // The parser behaves as if these standard mappings
156         // regardless of the actual map content:
157         //
158         //     "lt": "<",
159         //     "gt": ">",
160         //     "amp": "&",
161         //     "apos": "'",
162         //     "quot": `"`;
163         Entity map[string]string
164
165         // CharsetReader, if non-nil, defines a function to
166         // charset-conversion readers, converting from the p
167         // non-UTF-8 charset into UTF-8. If CharsetReader is
168         // returns an error, parsing stops with an error. On
169         // the CharsetReader's result values must be non-nil
170         CharsetReader func(charset string, input io.Reader)
171
172         r         io.ByteReader
173         buf       bytes.Buffer
174         saved     *bytes.Buffer
175         stk       *stack
176         free      *stack
177         needClose bool
178         toClose   Name
179         nextToken Token
180         nextByte  int
181         ns        map[string]string
182         err        error
183         line      int
184         tmp       [32]byte
185     }
186
187     // NewDecoder creates a new XML parser reading from r.
188     func NewDecoder(r io.Reader) *Decoder {
189         d := &Decoder{

```

```

190             ns:      make(map[string]string),
191             nextByte: -1,
192             line:    1,
193             Strict:  true,
194         }
195         d.switchToReader(r)
196         return d
197     }
198
199     // Token returns the next XML token in the input stream.
200     // At the end of the input stream, Token returns nil, io.EOF
201     //
202     // Slices of bytes in the returned token data refer to the
203     // parser's internal buffer and remain valid only until the
204     // call to Token. To acquire a copy of the bytes, call Copy
205     // or the token's Copy method.
206     //
207     // Token expands self-closing elements such as <br/>
208     // into separate start and end elements returned by successi
209     //
210     // Token guarantees that the StartElement and EndElement
211     // tokens it returns are properly nested and matched:
212     // if Token encounters an unexpected end element,
213     // it will return an error.
214     //
215     // Token implements XML name spaces as described by
216     // http://www.w3.org/TR/REC-xml-names/. Each of the
217     // Name structures contained in the Token has the Space
218     // set to the URL identifying its name space when known.
219     // If Token encounters an unrecognized name space prefix,
220     // it uses the prefix as the Space rather than report an err
221     func (d *Decoder) Token() (t Token, err error) {
222         if d.nextToken != nil {
223             t = d.nextToken
224             d.nextToken = nil
225         } else if t, err = d.RawToken(); err != nil {
226             return
227         }
228
229         if !d.Strict {
230             if t1, ok := d.autoClose(t); ok {
231                 d.nextToken = t
232                 t = t1
233             }
234         }
235         switch t1 := t.(type) {
236         case StartElement:
237             // In XML name spaces, the translations list
238             // attributes apply to the element name and
239             // to the other attribute names, so process

```

```

240         // the translations first.
241         for _, a := range t1.Attr {
242             if a.Name.Space == "xmlns" {
243                 v, ok := d.ns[a.Name.Local]
244                 d.pushNs(a.Name.Local, v, ok)
245                 d.ns[a.Name.Local] = a.Value
246             }
247             if a.Name.Space == "" && a.Name.Local == "xmlns" {
248                 // Default space for untagged
249                 v, ok := d.ns[""]
250                 d.pushNs("", v, ok)
251                 d.ns[""] = a.Value
252             }
253         }
254
255         d.translate(&t1.Name, true)
256         for i := range t1.Attr {
257             d.translate(&t1.Attr[i].Name, false)
258         }
259         d.pushElement(t1.Name)
260         t = t1
261     }
262     caseEndElement:
263         d.translate(&t1.Name, true)
264         if !d.popElement(&t1) {
265             return nil, d.err
266         }
267         t = t1
268     }
269     return
270 }
271
272 // Apply name space translation to name n.
273 // The default name space (for Space=="")
274 // applies only to element names, not to attribute names.
275 func (d *Decoder) translate(n *Name, isElementName bool) {
276     switch {
277     case n.Space == "xmlns":
278         return
279     case n.Space == "" && !isElementName:
280         return
281     case n.Space == "" && n.Local == "xmlns":
282         return
283     }
284     if v, ok := d.ns[n.Space]; ok {
285         n.Space = v
286     }
287 }
288

```

```

289 func (d *Decoder) switchToReader(r io.Reader) {
290     // Get efficient byte at a time reader.
291     // Assume that if reader has its own
292     // ReadByte, it's efficient enough.
293     // Otherwise, use bufio.
294     if rb, ok := r.(io.ByteReader); ok {
295         d.r = rb
296     } else {
297         d.r = bufio.NewReader(r)
298     }
299 }
300
301 // Parsing state - stack holds old name space translations
302 // and the current set of open elements. The translations t
303 // ending a given tag are *below* it on the stack, which is
304 // more work but forced on us by XML.
305 type stack struct {
306     next *stack
307     kind int
308     name Name
309     ok bool
310 }
311
312 const (
313     stkStart = iota
314     stkNs
315 )
316
317 func (d *Decoder) push(kind int) *stack {
318     s := d.free
319     if s != nil {
320         d.free = s.next
321     } else {
322         s = new(stack)
323     }
324     s.next = d.stk
325     s.kind = kind
326     d.stk = s
327     return s
328 }
329
330 func (d *Decoder) pop() *stack {
331     s := d.stk
332     if s != nil {
333         d.stk = s.next
334         s.next = d.free
335         d.free = s
336     }
337     return s

```

```

338 }
339
340 // Record that we are starting an element with the given name
341 func (d *Decoder) pushElement(name string) {
342     s := d.push(stkStart)
343     s.name = name
344 }
345
346 // Record that we are changing the value of ns[local].
347 // The old value is url, ok.
348 func (d *Decoder) pushNs(local string, url string, ok bool)
349     s := d.push(stkNs)
350     s.name.Local = local
351     s.name.Space = url
352     s.ok = ok
353 }
354
355 // Creates a SyntaxError with the current line number.
356 func (d *Decoder) syntaxError(msg string) error {
357     return &SyntaxError{Msg: msg, Line: d.line}
358 }
359
360 // Record that we are ending an element with the given name.
361 // The name must match the record at the top of the stack,
362 // which must be a pushElement record.
363 // After popping the element, apply any undo records from
364 // the stack to restore the name translations that existed
365 // before we saw this element.
366 func (d *Decoder) popElement(t *EndElement) bool {
367     s := d.pop()
368     name := t.Name
369     switch {
370     case s == nil || s.kind != stkStart:
371         d.err = d.syntaxError("unexpected end element")
372         return false
373     case s.name.Local != name.Local:
374         if !d.Strict {
375             d.needClose = true
376             d.toClose = t.Name
377             t.Name = s.name
378             return true
379         }
380         d.err = d.syntaxError("element <" + s.name.Local
381         return false
382     case s.name.Space != name.Space:
383         d.err = d.syntaxError("element <" + s.name.Local
384             "closed by </" + name.Local + "> in
385         return false
386     }
387 }

```

```

388         // Pop stack until a Start is on the top, undoing th
389         // translations that were associated with the elemen
390         for d.stk != nil && d.stk.kind != stkStart {
391             s := d.pop()
392             if s.ok {
393                 d.ns[s.name.Local] = s.name.Space
394             } else {
395                 delete(d.ns, s.name.Local)
396             }
397         }
398
399         return true
400     }
401
402     // If the top element on the stack is autoclosing and
403     // t is not the end tag, invent the end tag.
404     func (d *Decoder) autoClose(t Token) (Token, bool) {
405         if d.stk == nil || d.stk.kind != stkStart {
406             return nil, false
407         }
408         name := strings.ToLower(d.stk.name.Local)
409         for _, s := range d.AutoClose {
410             if strings.ToLower(s) == name {
411                 // This one should be auto closed if
412                 et, ok := t.(EndElement)
413                 if !ok || et.Name.Local != name {
414                     return EndElement{d.stk.name
415                 }
416                 break
417             }
418         }
419         return nil, false
420     }
421
422     // RawToken is like Token but does not verify that
423     // start and end elements match and does not translate
424     // name space prefixes to their corresponding URLs.
425     func (d *Decoder) RawToken() (Token, error) {
426         if d.err != nil {
427             return nil, d.err
428         }
429         if d.needClose {
430             // The last element we read was self-closing
431             // we returned just the StartElement half.
432             // Return the EndElement half now.
433             d.needClose = false
434             return EndElement{d.toClose}, nil
435         }
436

```

```

437     b, ok := d.getc()
438     if !ok {
439         return nil, d.err
440     }
441
442     if b != '<' {
443         // Text section.
444         d.ungetc(b)
445         data := d.text(-1, false)
446         if data == nil {
447             return nil, d.err
448         }
449         return CharData(data), nil
450     }
451
452     if b, ok = d.mustgetc(); !ok {
453         return nil, d.err
454     }
455     switch b {
456     case '/':
457         // </: End element
458         var name Name
459         if name, ok = d.nsname(); !ok {
460             if d.err == nil {
461                 d.err = d.syntaxError("expect
462             }
463             return nil, d.err
464         }
465         d.space()
466         if b, ok = d.mustgetc(); !ok {
467             return nil, d.err
468         }
469         if b != '>' {
470             d.err = d.syntaxError("invalid chara
471             return nil, d.err
472         }
473         return EndElement{name}, nil
474
475     case '?':
476         // <?: Processing instruction.
477         // TODO(rsc): Should parse the <?xml declara
478         // the version is 1.0 and the encoding is UT
479         var target string
480         if target, ok = d.name(); !ok {
481             if d.err == nil {
482                 d.err = d.syntaxError("expect
483             }
484             return nil, d.err
485         }

```

```

486         d.space()
487         d.buf.Reset()
488         var b0 byte
489         for {
490             if b, ok = d.mustgetc(); !ok {
491                 return nil, d.err
492             }
493             d.buf.WriteByte(b)
494             if b0 == '?' && b == '>' {
495                 break
496             }
497             b0 = b
498         }
499         data := d.buf.Bytes()
500         data = data[0 : len(data)-2] // chop ?>
501
502         if target == "xml" {
503             enc := procInstEncoding(string(data))
504             if enc != "" && enc != "utf-8" && en
505                 if d.CharsetReader == nil {
506                     d.err = fmt.Errorf("
507                         return nil, d.err
508                 }
509                 newr, err := d.CharsetReader
510                 if err != nil {
511                     d.err = fmt.Errorf("
512                         return nil, d.err
513                 }
514                 if newr == nil {
515                     panic("CharsetReader
516                 }
517                 d.switchToReader(newr)
518             }
519         }
520         return ProcInst{target, data}, nil
521
522     case '!':
523         // <!: Maybe comment, maybe CDATA.
524         if b, ok = d.mustgetc(); !ok {
525             return nil, d.err
526         }
527         switch b {
528         case '-': // <!--
529             // Probably <!-- for a comment.
530             if b, ok = d.mustgetc(); !ok {
531                 return nil, d.err
532             }
533             if b != '-' {
534                 d.err = d.syntaxError("inval
535                 return nil, d.err

```

```

536     }
537     // Look for terminator.
538     d.buf.Reset()
539     var b0, b1 byte
540     for {
541         if b, ok = d.mustgetc(); !ok
542             return nil, d.err
543     }
544     d.buf.WriteByte(b)
545     if b0 == '-' && b1 == '-' &&
546         break
547     }
548     b0, b1 = b1, b
549 }
550 data := d.buf.Bytes()
551 data = data[0 : len(data)-3] // chop
552 return Comment(data), nil
553
554 case '[': // 

```

```

585             break
586         }
587         d.buf.WriteByte(b)
588         switch {
589         case b == inquote:
590             inquote = 0
591
592         case inquote != 0:
593             // in quotes, no special act
594
595         case b == '\\' || b == '"':
596             inquote = b
597
598         case b == '>' && inquote == 0:
599             depth--
600
601         case b == '<' && inquote == 0:
602             depth++
603         }
604     }
605     return Directive(d.buf.Bytes()), nil
606 }
607
608 // Must be an open element like <a href="foo">
609 d.ungetc(b)
610
611 var (
612     name Name
613     empty bool
614     attr []Attr
615 )
616 if name, ok = d.nsname(); !ok {
617     if d.err == nil {
618         d.err = d.syntaxError("expected elem
619     }
620     return nil, d.err
621 }
622
623 attr = make([]Attr, 0, 4)
624 for {
625     d.space()
626     if b, ok = d.mustgetc(); !ok {
627         return nil, d.err
628     }
629     if b == '/' {
630         empty = true
631         if b, ok = d.mustgetc(); !ok {
632             return nil, d.err
633         }

```

```

634         if b != '>' {
635             d.err = d.syntaxError("expec
636             return nil, d.err
637         }
638         break
639     }
640     if b == '>' {
641         break
642     }
643     d.ungetc(b)
644
645     n := len(attr)
646     if n >= cap(attr) {
647         nattr := make([]Attr, n, 2*cap(attr))
648         copy(nattr, attr)
649         attr = nattr
650     }
651     attr = attr[0 : n+1]
652     a := &attr[n]
653     if a.Name, ok = d.nsname(); !ok {
654         if d.err == nil {
655             d.err = d.syntaxError("expec
656         }
657         return nil, d.err
658     }
659     d.space()
660     if b, ok = d.mustgetc(); !ok {
661         return nil, d.err
662     }
663     if b != '=' {
664         if d.Strict {
665             d.err = d.syntaxError("attri
666             return nil, d.err
667         } else {
668             d.ungetc(b)
669             a.Value = a.Name.Local
670         }
671     } else {
672         d.space()
673         data := d.attrval()
674         if data == nil {
675             return nil, d.err
676         }
677         a.Value = string(data)
678     }
679 }
680 if empty {
681     d.needClose = true
682     d.toClose = name
683 }

```

```

684         return StartElement{name, attr}, nil
685     }
686
687     func (d *Decoder) attrval() []byte {
688         b, ok := d.mustgetc()
689         if !ok {
690             return nil
691         }
692         // Handle quoted attribute values
693         if b == '"' || b == '\'' {
694             return d.text(int(b), false)
695         }
696         // Handle unquoted attribute values for strict parse
697         if d.Strict {
698             d.err = d.syntaxError("unquoted or missing a
699             return nil
700         }
701         // Handle unquoted attribute values for unstrict par
702         d.ungetc(b)
703         d.buf.Reset()
704         for {
705             b, ok = d.mustgetc()
706             if !ok {
707                 return nil
708             }
709             // http://www.w3.org/TR/REC-html40/intro/sgm
710             if 'a' <= b && b <= 'z' || 'A' <= b && b <=
711                 '0' <= b && b <= '9' || b == '_' ||
712                 d.buf.WriteByte(b)
713             } else {
714                 d.ungetc(b)
715                 break
716             }
717         }
718         return d.buf.Bytes()
719     }
720
721     // Skip spaces if any
722     func (d *Decoder) space() {
723         for {
724             b, ok := d.getc()
725             if !ok {
726                 return
727             }
728             switch b {
729             case ' ', '\r', '\n', '\t':
730                 default:
731                     d.ungetc(b)
732                     return

```

```

733         }
734     }
735 }
736
737 // Read a single byte.
738 // If there is no byte to read, return ok==false
739 // and leave the error in d.err.
740 // Maintain line number.
741 func (d *Decoder) getc() (b byte, ok bool) {
742     if d.err != nil {
743         return 0, false
744     }
745     if d.nextByte >= 0 {
746         b = byte(d.nextByte)
747         d.nextByte = -1
748     } else {
749         b, d.err = d.r.ReadByte()
750         if d.err != nil {
751             return 0, false
752         }
753         if d.saved != nil {
754             d.saved.WriteByte(b)
755         }
756     }
757     if b == '\n' {
758         d.line++
759     }
760     return b, true
761 }
762
763 // Return saved offset.
764 // If we did ungetc (nextByte >= 0), have to back up one.
765 func (d *Decoder) savedOffset() int {
766     n := d.saved.Len()
767     if d.nextByte >= 0 {
768         n--
769     }
770     return n
771 }
772
773 // Must read a single byte.
774 // If there is no byte to read,
775 // set d.err to SyntaxError("unexpected EOF")
776 // and return ok==false
777 func (d *Decoder) mustgetc() (b byte, ok bool) {
778     if b, ok = d.getc(); !ok {
779         if d.err == io.EOF {
780             d.err = d.syntaxError("unexpected EO
781         }

```

```

782         }
783         return
784     }
785
786     // Unread a single byte.
787     func (d *Decoder) ungetc(b byte) {
788         if b == '\n' {
789             d.line--
790         }
791         d.nextByte = int(b)
792     }
793
794     var entity = map[string]int{
795         "lt": '<',
796         "gt": '>',
797         "amp": '&',
798         "apos": '\'',
799         "quot": '"',
800     }
801
802     // Read plain text section (XML calls it character data).
803     // If quote >= 0, we are in a quoted string and need to find
804     // If cdata == true, we are in a  section and need
805     // On failure return nil and leave the error in d.err.
806     func (d *Decoder) text(quote int, cdata bool) []byte {
807         var b0, b1 byte
808         var trunc int
809         d.buf.Reset()
810     Input:
811         for {
812             b, ok := d.getc()
813             if !ok {
814                 if cdata {
815                     if d.err == io.EOF {
816                         d.err = d.syntaxError
817                     }
818                     return nil
819                 }
820                 break Input
821             }
822
823             // <![CDATA[ section ends with ]&gt;.
824             // It is an error for ]&gt; to appear in ordin
825             if b0 == ']' &amp;&amp; b1 == ']' &amp;&amp; b == '&gt;' {
826                 if cdata {
827                     trunc = 2
828                     break Input
829                 }
830                 d.err = d.syntaxError("unescaped ]&gt;")
831                 return nil
</pre>
</div>
```

```

832     }
833
834     // Stop reading text if we see a <.
835     if b == '<' && !cdata {
836         if quote >= 0 {
837             d.err = d.syntaxError("unesc
838             return nil
839         }
840         d.ungetc('<')
841         break Input
842     }
843     if quote >= 0 && b == byte(quote) {
844         break Input
845     }
846     if b == '&' && !cdata {
847         // Read escaped character expression
848         // XML in all its glory allows a doc
849         // its own character names with <!EN
850         // Parsers are required to recognize
851         // even if they have not been declar
852         var i int
853         for i = 0; i < len(d.tmp); i++ {
854             var ok bool
855             d.tmp[i], ok = d.getc()
856             if !ok {
857                 if d.err == io.EOF {
858                     d.err = d.sy
859                 }
860                 return nil
861             }
862             c := d.tmp[i]
863             if c == ';' {
864                 break
865             }
866             if 'a' <= c && c <= 'z' ||
867                'A' <= c && c <= 'Z'
868                '0' <= c && c <= '9'
869                c == '_' || c == '#'
870                continue
871         }
872         d.ungetc(c)
873         break
874     }
875     s := string(d.tmp[0:i])
876     if i >= len(d.tmp) {
877         if !d.Strict {
878             b0, b1 = 0, 0
879             d.buf.WriteByte('&')
880             d.buf.Write(d.tmp[0:

```

```

881             continue Input
882         }
883         d.err = d.syntaxError("chara
884         return nil
885     }
886     var haveText bool
887     var text string
888     if i >= 2 && s[0] == '#' {
889         var n uint64
890         var err error
891         if i >= 3 && s[1] == 'x' {
892             n, err = strconv.Par
893         } else {
894             n, err = strconv.Par
895         }
896         if err == nil && n <= unicond
897             text = string(n)
898             haveText = true
899     }
900     } else {
901         if r, ok := entity[s]; ok {
902             text = string(r)
903             haveText = true
904         } else if d.Entity != nil {
905             text, haveText = d.E
906         }
907     }
908     if !haveText {
909         if !d.Strict {
910             b0, b1 = 0, 0
911             d.buf.WriteByte('&')
912             d.buf.Write(d.tmp[0:
913             continue Input
914         }
915         d.err = d.syntaxError("inval
916         return nil
917     }
918     d.buf.Write([]byte(text))
919     b0, b1 = 0, 0
920     continue Input
921 }
922     d.buf.WriteByte(b)
923     b0, b1 = b1, b
924 }
925 data := d.buf.Bytes()
926 data = data[0 : len(data)-trunc]
927
928 // Inspect each rune for being a disallowed characte
929 buf := data

```

```

930     for len(buf) > 0 {
931         r, size := utf8.DecodeRune(buf)
932         if r == utf8.RuneError && size == 1 {
933             d.err = d.syntaxError("invalid UTF-8")
934             return nil
935         }
936         buf = buf[size:]
937         if !isInCharacterRange(r) {
938             d.err = d.syntaxError(fmt.Sprintf("i
939             return nil
940         }
941     }
942
943     // Must rewrite \r and \r\n into \n.
944     w := 0
945     for r := 0; r < len(data); r++ {
946         b := data[r]
947         if b == '\r' {
948             if r+1 < len(data) && data[r+1] == '
949                 continue
950             }
951             b = '\n'
952         }
953         data[w] = b
954         w++
955     }
956     return data[0:w]
957 }
958
959 // Decide whether the given rune is in the XML Character Ran
960 // the Char production of http://www.xml.com/axml/testaxml.h
961 // Section 2.2 Characters.
962 func isInCharacterRange(r rune) (inrange bool) {
963     return r == 0x09 ||
964         r == 0x0A ||
965         r == 0x0D ||
966         r >= 0x20 && r <= 0xDF77 ||
967         r >= 0xE000 && r <= 0xFFFFD ||
968         r >= 0x10000 && r <= 0x10FFFF
969 }
970
971 // Get name space name: name with a : stuck in the middle.
972 // The part before the : is the name space identifier.
973 func (d *Decoder) nsname() (name Name, ok bool) {
974     s, ok := d.name()
975     if !ok {
976         return
977     }
978     i := strings.Index(s, ":")
979     if i < 0 {

```

```

980         name.Local = s
981     } else {
982         name.Space = s[0:i]
983         name.Local = s[i+1:]
984     }
985     return name, true
986 }
987
988 // Get name: /first(first|second)*/
989 // Do not set d.err if the name is missing (unless unexpecte
990 // let the caller provide better context.
991 func (d *Decoder) name() (s string, ok bool) {
992     var b byte
993     if b, ok = d.mustgetc(); !ok {
994         return
995     }
996
997     // As a first approximation, we gather the bytes [A-
998     if b < utf8.RuneSelf && !isNameByte(b) {
999         d.ungetc(b)
1000        return "", false
1001    }
1002    d.buf.Reset()
1003    d.buf.WriteByte(b)
1004    for {
1005        if b, ok = d.mustgetc(); !ok {
1006            return
1007        }
1008        if b < utf8.RuneSelf && !isNameByte(b) {
1009            d.ungetc(b)
1010            break
1011        }
1012        d.buf.WriteByte(b)
1013    }
1014
1015    // Then we check the characters.
1016    s = d.buf.String()
1017    for i, c := range s {
1018        if !unicode.Is(first, c) && (i == 0 || !unic
1019            d.err = d.syntaxError("invalid XML n
1020            return "", false
1021        }
1022    }
1023    return s, true
1024 }
1025
1026 func isNameByte(c byte) bool {
1027     return 'A' <= c && c <= 'Z' ||
1028         'a' <= c && c <= 'z' ||

```

```

1029         '0' <= c && c <= '9' ||
1030         c == '_' || c == ':' || c == '.' || c == '-'
1031     }
1032
1033 // These tables were generated by cut and paste from Appendi
1034 // the XML spec at http://www.xml.com/axml/testaxml.htm
1035 // and then reformatting. First corresponds to (Letter | '_
1036 // and second corresponds to NameChar.
1037
1038 var first = &unicode.RangeTable{
1039     R16: []unicode.Range16{
1040         {0x003A, 0x003A, 1},
1041         {0x0041, 0x005A, 1},
1042         {0x005F, 0x005F, 1},
1043         {0x0061, 0x007A, 1},
1044         {0x00C0, 0x00D6, 1},
1045         {0x00D8, 0x00F6, 1},
1046         {0x00F8, 0x00FF, 1},
1047         {0x0100, 0x0131, 1},
1048         {0x0134, 0x013E, 1},
1049         {0x0141, 0x0148, 1},
1050         {0x014A, 0x017E, 1},
1051         {0x0180, 0x01C3, 1},
1052         {0x01CD, 0x01F0, 1},
1053         {0x01F4, 0x01F5, 1},
1054         {0x01FA, 0x0217, 1},
1055         {0x0250, 0x02A8, 1},
1056         {0x02BB, 0x02C1, 1},
1057         {0x0386, 0x0386, 1},
1058         {0x0388, 0x038A, 1},
1059         {0x038C, 0x038C, 1},
1060         {0x038E, 0x03A1, 1},
1061         {0x03A3, 0x03CE, 1},
1062         {0x03D0, 0x03D6, 1},
1063         {0x03DA, 0x03E0, 2},
1064         {0x03E2, 0x03F3, 1},
1065         {0x0401, 0x040C, 1},
1066         {0x040E, 0x044F, 1},
1067         {0x0451, 0x045C, 1},
1068         {0x045E, 0x0481, 1},
1069         {0x0490, 0x04C4, 1},
1070         {0x04C7, 0x04C8, 1},
1071         {0x04CB, 0x04CC, 1},
1072         {0x04D0, 0x04EB, 1},
1073         {0x04EE, 0x04F5, 1},
1074         {0x04F8, 0x04F9, 1},
1075         {0x0531, 0x0556, 1},
1076         {0x0559, 0x0559, 1},
1077         {0x0561, 0x0586, 1},

```

1078 {0x05D0, 0x05EA, 1},
1079 {0x05F0, 0x05F2, 1},
1080 {0x0621, 0x063A, 1},
1081 {0x0641, 0x064A, 1},
1082 {0x0671, 0x06B7, 1},
1083 {0x06BA, 0x06BE, 1},
1084 {0x06C0, 0x06CE, 1},
1085 {0x06D0, 0x06D3, 1},
1086 {0x06D5, 0x06D5, 1},
1087 {0x06E5, 0x06E6, 1},
1088 {0x0905, 0x0939, 1},
1089 {0x093D, 0x093D, 1},
1090 {0x0958, 0x0961, 1},
1091 {0x0985, 0x098C, 1},
1092 {0x098F, 0x0990, 1},
1093 {0x0993, 0x09A8, 1},
1094 {0x09AA, 0x09B0, 1},
1095 {0x09B2, 0x09B2, 1},
1096 {0x09B6, 0x09B9, 1},
1097 {0x09DC, 0x09DD, 1},
1098 {0x09DF, 0x09E1, 1},
1099 {0x09F0, 0x09F1, 1},
1100 {0x0A05, 0x0A0A, 1},
1101 {0x0A0F, 0x0A10, 1},
1102 {0x0A13, 0x0A28, 1},
1103 {0x0A2A, 0x0A30, 1},
1104 {0x0A32, 0x0A33, 1},
1105 {0x0A35, 0x0A36, 1},
1106 {0x0A38, 0x0A39, 1},
1107 {0x0A59, 0x0A5C, 1},
1108 {0x0A5E, 0x0A5E, 1},
1109 {0x0A72, 0x0A74, 1},
1110 {0x0A85, 0x0A8B, 1},
1111 {0x0A8D, 0x0A8D, 1},
1112 {0x0A8F, 0x0A91, 1},
1113 {0x0A93, 0x0AA8, 1},
1114 {0x0AAA, 0x0AB0, 1},
1115 {0x0AB2, 0x0AB3, 1},
1116 {0x0AB5, 0x0AB9, 1},
1117 {0x0ABD, 0x0AE0, 0x23},
1118 {0x0B05, 0x0B0C, 1},
1119 {0x0B0F, 0x0B10, 1},
1120 {0x0B13, 0x0B28, 1},
1121 {0x0B2A, 0x0B30, 1},
1122 {0x0B32, 0x0B33, 1},
1123 {0x0B36, 0x0B39, 1},
1124 {0x0B3D, 0x0B3D, 1},
1125 {0x0B5C, 0x0B5D, 1},
1126 {0x0B5F, 0x0B61, 1},
1127 {0x0B85, 0x0B8A, 1},

1128	{0x0B8E, 0x0B90, 1},
1129	{0x0B92, 0x0B95, 1},
1130	{0x0B99, 0x0B9A, 1},
1131	{0x0B9C, 0x0B9C, 1},
1132	{0x0B9E, 0x0B9F, 1},
1133	{0x0BA3, 0x0BA4, 1},
1134	{0x0BA8, 0x0BAA, 1},
1135	{0x0BAE, 0x0BB5, 1},
1136	{0x0BB7, 0x0BB9, 1},
1137	{0x0C05, 0x0C0C, 1},
1138	{0x0C0E, 0x0C10, 1},
1139	{0x0C12, 0x0C28, 1},
1140	{0x0C2A, 0x0C33, 1},
1141	{0x0C35, 0x0C39, 1},
1142	{0x0C60, 0x0C61, 1},
1143	{0x0C85, 0x0C8C, 1},
1144	{0x0C8E, 0x0C90, 1},
1145	{0x0C92, 0x0CA8, 1},
1146	{0x0CAA, 0x0CB3, 1},
1147	{0x0CB5, 0x0CB9, 1},
1148	{0x0CDE, 0x0CDE, 1},
1149	{0x0CE0, 0x0CE1, 1},
1150	{0x0D05, 0x0D0C, 1},
1151	{0x0D0E, 0x0D10, 1},
1152	{0x0D12, 0x0D28, 1},
1153	{0x0D2A, 0x0D39, 1},
1154	{0x0D60, 0x0D61, 1},
1155	{0x0E01, 0x0E2E, 1},
1156	{0x0E30, 0x0E30, 1},
1157	{0x0E32, 0x0E33, 1},
1158	{0x0E40, 0x0E45, 1},
1159	{0x0E81, 0x0E82, 1},
1160	{0x0E84, 0x0E84, 1},
1161	{0x0E87, 0x0E88, 1},
1162	{0x0E8A, 0x0E8D, 3},
1163	{0x0E94, 0x0E97, 1},
1164	{0x0E99, 0x0E9F, 1},
1165	{0x0EA1, 0x0EA3, 1},
1166	{0x0EA5, 0x0EA7, 2},
1167	{0x0EAA, 0x0EAB, 1},
1168	{0x0EAD, 0x0EAE, 1},
1169	{0x0EB0, 0x0EB0, 1},
1170	{0x0EB2, 0x0EB3, 1},
1171	{0x0EBD, 0x0EBD, 1},
1172	{0x0EC0, 0x0EC4, 1},
1173	{0x0F40, 0x0F47, 1},
1174	{0x0F49, 0x0F69, 1},
1175	{0x10A0, 0x10C5, 1},
1176	{0x10D0, 0x10F6, 1},

1177 {0x1100, 0x1100, 1},
1178 {0x1102, 0x1103, 1},
1179 {0x1105, 0x1107, 1},
1180 {0x1109, 0x1109, 1},
1181 {0x110B, 0x110C, 1},
1182 {0x110E, 0x1112, 1},
1183 {0x113C, 0x1140, 2},
1184 {0x114C, 0x1150, 2},
1185 {0x1154, 0x1155, 1},
1186 {0x1159, 0x1159, 1},
1187 {0x115F, 0x1161, 1},
1188 {0x1163, 0x1169, 2},
1189 {0x116D, 0x116E, 1},
1190 {0x1172, 0x1173, 1},
1191 {0x1175, 0x119E, 0x119E - 0x1175},
1192 {0x11A8, 0x11AB, 0x11AB - 0x11A8},
1193 {0x11AE, 0x11AF, 1},
1194 {0x11B7, 0x11B8, 1},
1195 {0x11BA, 0x11BA, 1},
1196 {0x11BC, 0x11C2, 1},
1197 {0x11EB, 0x11F0, 0x11F0 - 0x11EB},
1198 {0x11F9, 0x11F9, 1},
1199 {0x1E00, 0x1E9B, 1},
1200 {0x1EA0, 0x1EF9, 1},
1201 {0x1F00, 0x1F15, 1},
1202 {0x1F18, 0x1F1D, 1},
1203 {0x1F20, 0x1F45, 1},
1204 {0x1F48, 0x1F4D, 1},
1205 {0x1F50, 0x1F57, 1},
1206 {0x1F59, 0x1F5B, 0x1F5B - 0x1F59},
1207 {0x1F5D, 0x1F5D, 1},
1208 {0x1F5F, 0x1F7D, 1},
1209 {0x1F80, 0x1FB4, 1},
1210 {0x1FB6, 0x1FBC, 1},
1211 {0x1FBE, 0x1FBE, 1},
1212 {0x1FC2, 0x1FC4, 1},
1213 {0x1FC6, 0x1FCC, 1},
1214 {0x1FD0, 0x1FD3, 1},
1215 {0x1FD6, 0x1FDB, 1},
1216 {0x1FE0, 0x1FEC, 1},
1217 {0x1FF2, 0x1FF4, 1},
1218 {0x1FF6, 0x1FFC, 1},
1219 {0x2126, 0x2126, 1},
1220 {0x212A, 0x212B, 1},
1221 {0x212E, 0x212E, 1},
1222 {0x2180, 0x2182, 1},
1223 {0x3007, 0x3007, 1},
1224 {0x3021, 0x3029, 1},
1225 {0x3041, 0x3094, 1},

```

1226             {0x30A1, 0x30FA, 1},
1227             {0x3105, 0x312C, 1},
1228             {0x4E00, 0x9FA5, 1},
1229             {0xAC00, 0xD7A3, 1},
1230         },
1231     }
1232
1233     var second = &unicode.RangeTable{
1234         R16: []unicode.Range16{
1235             {0x002D, 0x002E, 1},
1236             {0x0030, 0x0039, 1},
1237             {0x00B7, 0x00B7, 1},
1238             {0x02D0, 0x02D1, 1},
1239             {0x0300, 0x0345, 1},
1240             {0x0360, 0x0361, 1},
1241             {0x0387, 0x0387, 1},
1242             {0x0483, 0x0486, 1},
1243             {0x0591, 0x05A1, 1},
1244             {0x05A3, 0x05B9, 1},
1245             {0x05BB, 0x05BD, 1},
1246             {0x05BF, 0x05BF, 1},
1247             {0x05C1, 0x05C2, 1},
1248             {0x05C4, 0x0640, 0x0640 - 0x05C4},
1249             {0x064B, 0x0652, 1},
1250             {0x0660, 0x0669, 1},
1251             {0x0670, 0x0670, 1},
1252             {0x06D6, 0x06DC, 1},
1253             {0x06DD, 0x06DF, 1},
1254             {0x06E0, 0x06E4, 1},
1255             {0x06E7, 0x06E8, 1},
1256             {0x06EA, 0x06ED, 1},
1257             {0x06F0, 0x06F9, 1},
1258             {0x0901, 0x0903, 1},
1259             {0x093C, 0x093C, 1},
1260             {0x093E, 0x094C, 1},
1261             {0x094D, 0x094D, 1},
1262             {0x0951, 0x0954, 1},
1263             {0x0962, 0x0963, 1},
1264             {0x0966, 0x096F, 1},
1265             {0x0981, 0x0983, 1},
1266             {0x09BC, 0x09BC, 1},
1267             {0x09BE, 0x09BF, 1},
1268             {0x09C0, 0x09C4, 1},
1269             {0x09C7, 0x09C8, 1},
1270             {0x09CB, 0x09CD, 1},
1271             {0x09D7, 0x09D7, 1},
1272             {0x09E2, 0x09E3, 1},
1273             {0x09E6, 0x09EF, 1},
1274             {0x0A02, 0x0A3C, 0x3A},
1275             {0x0A3E, 0x0A3F, 1},

```

1276	{0x0A40, 0x0A42, 1},
1277	{0x0A47, 0x0A48, 1},
1278	{0x0A4B, 0x0A4D, 1},
1279	{0x0A66, 0x0A6F, 1},
1280	{0x0A70, 0x0A71, 1},
1281	{0x0A81, 0x0A83, 1},
1282	{0x0ABC, 0x0ABC, 1},
1283	{0x0ABE, 0x0AC5, 1},
1284	{0x0AC7, 0x0AC9, 1},
1285	{0x0ACB, 0x0ACD, 1},
1286	{0x0AE6, 0x0AEF, 1},
1287	{0x0B01, 0x0B03, 1},
1288	{0x0B3C, 0x0B3C, 1},
1289	{0x0B3E, 0x0B43, 1},
1290	{0x0B47, 0x0B48, 1},
1291	{0x0B4B, 0x0B4D, 1},
1292	{0x0B56, 0x0B57, 1},
1293	{0x0B66, 0x0B6F, 1},
1294	{0x0B82, 0x0B83, 1},
1295	{0x0BBE, 0x0BC2, 1},
1296	{0x0BC6, 0x0BC8, 1},
1297	{0x0BCA, 0x0BCD, 1},
1298	{0x0BD7, 0x0BD7, 1},
1299	{0x0BE7, 0x0BEF, 1},
1300	{0x0C01, 0x0C03, 1},
1301	{0x0C3E, 0x0C44, 1},
1302	{0x0C46, 0x0C48, 1},
1303	{0x0C4A, 0x0C4D, 1},
1304	{0x0C55, 0x0C56, 1},
1305	{0x0C66, 0x0C6F, 1},
1306	{0x0C82, 0x0C83, 1},
1307	{0x0CBE, 0x0CC4, 1},
1308	{0x0CC6, 0x0CC8, 1},
1309	{0x0CCA, 0x0CCD, 1},
1310	{0x0CD5, 0x0CD6, 1},
1311	{0x0CE6, 0x0CEF, 1},
1312	{0x0D02, 0x0D03, 1},
1313	{0x0D3E, 0x0D43, 1},
1314	{0x0D46, 0x0D48, 1},
1315	{0x0D4A, 0x0D4D, 1},
1316	{0x0D57, 0x0D57, 1},
1317	{0x0D66, 0x0D6F, 1},
1318	{0x0E31, 0x0E31, 1},
1319	{0x0E34, 0x0E3A, 1},
1320	{0x0E46, 0x0E46, 1},
1321	{0x0E47, 0x0E4E, 1},
1322	{0x0E50, 0x0E59, 1},
1323	{0x0EB1, 0x0EB1, 1},
1324	{0x0EB4, 0x0EB9, 1},

```

1325         {0x0EBB, 0x0EBC, 1},
1326         {0x0EC6, 0x0EC6, 1},
1327         {0x0EC8, 0x0ECD, 1},
1328         {0x0ED0, 0x0ED9, 1},
1329         {0x0F18, 0x0F19, 1},
1330         {0x0F20, 0x0F29, 1},
1331         {0x0F35, 0x0F39, 2},
1332         {0x0F3E, 0x0F3F, 1},
1333         {0x0F71, 0x0F84, 1},
1334         {0x0F86, 0x0F8B, 1},
1335         {0x0F90, 0x0F95, 1},
1336         {0x0F97, 0x0F97, 1},
1337         {0x0F99, 0x0FAD, 1},
1338         {0x0FB1, 0x0FB7, 1},
1339         {0x0FB9, 0x0FB9, 1},
1340         {0x20D0, 0x20DC, 1},
1341         {0x20E1, 0x3005, 0x3005 - 0x20E1},
1342         {0x302A, 0x302F, 1},
1343         {0x3031, 0x3035, 1},
1344         {0x3099, 0x309A, 1},
1345         {0x309D, 0x309E, 1},
1346         {0x30FC, 0x30FE, 1},
1347     },
1348 }
1349
1350 // HTMLEntity is an entity map containing translations for t
1351 // standard HTML entity characters.
1352 var HTMLEntity = htmlEntity
1353
1354 var htmlEntity = map[string]string{
1355     /*
1356         hget http://www.w3.org/TR/html4/sgml/entities
1357         ssam '
1358             ,y /\&gt;/ x/\&lt;(.|\n)+/ s/\n/ /g
1359             ,x v/^\&lt;!ENTITY/d
1360             ,s/\&lt;!ENTITY ([^ ]+) .*U\+([0-9A-
1361         '
1362     */
1363     "nbsp":    "\u00A0",
1364     "iexcl":  "\u00A1",
1365     "cent":   "\u00A2",
1366     "pound":  "\u00A3",
1367     "curren": "\u00A4",
1368     "yen":    "\u00A5",
1369     "brvbar": "\u00A6",
1370     "sect":   "\u00A7",
1371     "uml":    "\u00A8",
1372     "copy":   "\u00A9",
1373     "ordf":   "\u00AA",

```

1374 "laquo": "\u00AB",
1375 "not": "\u00AC",
1376 "shy": "\u00AD",
1377 "reg": "\u00AE",
1378 "macr": "\u00AF",
1379 "deg": "\u00B0",
1380 "plusmn": "\u00B1",
1381 "sup2": "\u00B2",
1382 "sup3": "\u00B3",
1383 "acute": "\u00B4",
1384 "micro": "\u00B5",
1385 "para": "\u00B6",
1386 "middot": "\u00B7",
1387 "cedil": "\u00B8",
1388 "sup1": "\u00B9",
1389 "ordm": "\u00BA",
1390 "raquo": "\u00BB",
1391 "frac14": "\u00BC",
1392 "frac12": "\u00BD",
1393 "frac34": "\u00BE",
1394 "iquest": "\u00BF",
1395 "Agrave": "\u00C0",
1396 "Aacute": "\u00C1",
1397 "Acirc": "\u00C2",
1398 "Atilde": "\u00C3",
1399 "Auml": "\u00C4",
1400 "Aring": "\u00C5",
1401 "AElig": "\u00C6",
1402 "Ccedil": "\u00C7",
1403 "Egrave": "\u00C8",
1404 "Eacute": "\u00C9",
1405 "Ecirc": "\u00CA",
1406 "Euml": "\u00CB",
1407 "Igrave": "\u00CC",
1408 "Iacute": "\u00CD",
1409 "Icirc": "\u00CE",
1410 "Iuml": "\u00CF",
1411 "ETH": "\u00D0",
1412 "Ntilde": "\u00D1",
1413 "Ograve": "\u00D2",
1414 "Oacute": "\u00D3",
1415 "Ocirc": "\u00D4",
1416 "Otilde": "\u00D5",
1417 "Ouml": "\u00D6",
1418 "times": "\u00D7",
1419 "Oslash": "\u00D8",
1420 "Ugrave": "\u00D9",
1421 "Uacute": "\u00DA",
1422 "Ucirc": "\u00DB",
1423 "Uuml": "\u00DC",

1424	"Yacute":	"\u00DD",
1425	"THORN":	"\u00DE",
1426	"szlig":	"\u00DF",
1427	"agrave":	"\u00E0",
1428	"aacute":	"\u00E1",
1429	"acirc":	"\u00E2",
1430	"atilde":	"\u00E3",
1431	"auml":	"\u00E4",
1432	"aring":	"\u00E5",
1433	"aelig":	"\u00E6",
1434	"ccedil":	"\u00E7",
1435	"egrave":	"\u00E8",
1436	"eacute":	"\u00E9",
1437	"ecirc":	"\u00EA",
1438	"euml":	"\u00EB",
1439	"igrave":	"\u00EC",
1440	"iacute":	"\u00ED",
1441	"icirc":	"\u00EE",
1442	"iuml":	"\u00EF",
1443	"eth":	"\u00F0",
1444	"ntilde":	"\u00F1",
1445	"ograve":	"\u00F2",
1446	"oacute":	"\u00F3",
1447	"ocirc":	"\u00F4",
1448	"otilde":	"\u00F5",
1449	"ouml":	"\u00F6",
1450	"divide":	"\u00F7",
1451	"oslash":	"\u00F8",
1452	"ugrave":	"\u00F9",
1453	"uacute":	"\u00FA",
1454	"ucirc":	"\u00FB",
1455	"uuml":	"\u00FC",
1456	"yacute":	"\u00FD",
1457	"thorn":	"\u00FE",
1458	"yuml":	"\u00FF",
1459	"fnof":	"\u0192",
1460	"Alpha":	"\u0391",
1461	"Beta":	"\u0392",
1462	"Gamma":	"\u0393",
1463	"Delta":	"\u0394",
1464	"Epsilon":	"\u0395",
1465	"Zeta":	"\u0396",
1466	"Eta":	"\u0397",
1467	"Theta":	"\u0398",
1468	"Iota":	"\u0399",
1469	"Kappa":	"\u039A",
1470	"Lambda":	"\u039B",
1471	"Mu":	"\u039C",
1472	"Nu":	"\u039D",

1473 "Xi": "\u039E",
1474 "Omicron": "\u039F",
1475 "Pi": "\u03A0",
1476 "Rho": "\u03A1",
1477 "Sigma": "\u03A3",
1478 "Tau": "\u03A4",
1479 "Upsilon": "\u03A5",
1480 "Phi": "\u03A6",
1481 "Chi": "\u03A7",
1482 "Psi": "\u03A8",
1483 "Omega": "\u03A9",
1484 "alpha": "\u03B1",
1485 "beta": "\u03B2",
1486 "gamma": "\u03B3",
1487 "delta": "\u03B4",
1488 "epsilon": "\u03B5",
1489 "zeta": "\u03B6",
1490 "eta": "\u03B7",
1491 "theta": "\u03B8",
1492 "iota": "\u03B9",
1493 "kappa": "\u03BA",
1494 "lambda": "\u03BB",
1495 "mu": "\u03BC",
1496 "nu": "\u03BD",
1497 "xi": "\u03BE",
1498 "omicron": "\u03BF",
1499 "pi": "\u03C0",
1500 "rho": "\u03C1",
1501 "sigmaf": "\u03C2",
1502 "sigma": "\u03C3",
1503 "tau": "\u03C4",
1504 "upsilon": "\u03C5",
1505 "phi": "\u03C6",
1506 "chi": "\u03C7",
1507 "psi": "\u03C8",
1508 "omega": "\u03C9",
1509 "thetasym": "\u03D1",
1510 "upsih": "\u03D2",
1511 "piv": "\u03D6",
1512 "bull": "\u2022",
1513 "hellip": "\u2026",
1514 "prime": "\u2032",
1515 "Prime": "\u2033",
1516 "oline": "\u203E",
1517 "frac1": "\u2044",
1518 "weierp": "\u2118",
1519 "image": "\u2111",
1520 "real": "\u211C",
1521 "trade": "\u2122",

1522 "alefsym": "\u2135",
1523 "larr": "\u2190",
1524 "uarr": "\u2191",
1525 "rarr": "\u2192",
1526 "darr": "\u2193",
1527 "harr": "\u2194",
1528 "crarr": "\u21B5",
1529 "lArr": "\u21D0",
1530 "uArr": "\u21D1",
1531 "rArr": "\u21D2",
1532 "dArr": "\u21D3",
1533 "hArr": "\u21D4",
1534 "forall": "\u2200",
1535 "part": "\u2202",
1536 "exist": "\u2203",
1537 "empty": "\u2205",
1538 "nabla": "\u2207",
1539 "isin": "\u2208",
1540 "notin": "\u2209",
1541 "ni": "\u220B",
1542 "prod": "\u220F",
1543 "sum": "\u2211",
1544 "minus": "\u2212",
1545 "lowast": "\u2217",
1546 "radic": "\u221A",
1547 "prop": "\u221D",
1548 "infin": "\u221E",
1549 "ang": "\u2220",
1550 "and": "\u2227",
1551 "or": "\u2228",
1552 "cap": "\u2229",
1553 "cup": "\u222A",
1554 "int": "\u222B",
1555 "there4": "\u2234",
1556 "sim": "\u223C",
1557 "cong": "\u2245",
1558 "asymp": "\u2248",
1559 "ne": "\u2260",
1560 "equiv": "\u2261",
1561 "le": "\u2264",
1562 "ge": "\u2265",
1563 "sub": "\u2282",
1564 "sup": "\u2283",
1565 "nsub": "\u2284",
1566 "sube": "\u2286",
1567 "supe": "\u2287",
1568 "oplus": "\u2295",
1569 "otimes": "\u2297",
1570 "perp": "\u22A5",
1571 "sdot": "\u22C5",

```

1572         "lceil":      "\u2308",
1573         "rceil":      "\u2309",
1574         "lfloor":     "\u230A",
1575         "rfloor":     "\u230B",
1576         "lang":       "\u2329",
1577         "rang":       "\u232A",
1578         "loz":        "\u25CA",
1579         "spades":     "\u2660",
1580         "clubs":      "\u2663",
1581         "hearts":     "\u2665",
1582         "diams":     "\u2666",
1583         "quot":       "\u0022",
1584         "amp":        "\u0026",
1585         "lt":         "\u003C",
1586         "gt":         "\u003E",
1587         "OElig":      "\u0152",
1588         "oelig":      "\u0153",
1589         "Scaron":     "\u0160",
1590         "scaron":     "\u0161",
1591         "Yuml":       "\u0178",
1592         "circ":       "\u02C6",
1593         "tilde":      "\u02DC",
1594         "ensp":       "\u2002",
1595         "emsp":       "\u2003",
1596         "thinsp":     "\u2009",
1597         "zwnj":       "\u200C",
1598         "zwj":        "\u200D",
1599         "lrm":        "\u200E",
1600         "rlm":        "\u200F",
1601         "ndash":      "\u2013",
1602         "mdash":      "\u2014",
1603         "lsquo":      "\u2018",
1604         "rsquo":      "\u2019",
1605         "sbquo":      "\u201A",
1606         "ldquo":      "\u201C",
1607         "rdquo":      "\u201D",
1608         "bdquo":      "\u201E",
1609         "dagger":     "\u2020",
1610         "Dagger":     "\u2021",
1611         "permil":     "\u2030",
1612         "lsaquo":     "\u2039",
1613         "rsaquo":     "\u203A",
1614         "euro":       "\u20AC",
1615     }
1616
1617     // HTMLAutoClose is the set of HTML elements that
1618     // should be considered to close automatically.
1619     var HTMLAutoClose = htmlAutoClose
1620

```

```

1621 var htmlAutoClose = []string{
1622     /*
1623         hget http://www.w3.org/TR/html4/loose.dtd |
1624         9 sed -n 's/<!ELEMENT (.*) - O EMPTY.+ / "\1"
1625     */
1626     "basefont",
1627     "br",
1628     "area",
1629     "link",
1630     "img",
1631     "param",
1632     "hr",
1633     "input",
1634     "col",
1635     "frame",
1636     "isindex",
1637     "base",
1638     "meta",
1639 }
1640
1641 var (
1642     esc_quot = []byte("&#34;") // shorter than "&quot;"
1643     esc_apos = []byte("&#39;") // shorter than "&apos;"
1644     esc_amp  = []byte("&amp;")
1645     esc_lt   = []byte("&lt;")
1646     esc_gt   = []byte("&gt;")
1647 )
1648
1649 // Escape writes to w the properly escaped XML equivalent
1650 // of the plain text data s.
1651 func Escape(w io.Writer, s []byte) {
1652     var esc []byte
1653     last := 0
1654     for i, c := range s {
1655         switch c {
1656             case '"':
1657                 esc = esc_quot
1658             case '\'':
1659                 esc = esc_apos
1660             case '&':
1661                 esc = esc_amp
1662             case '<':
1663                 esc = esc_lt
1664             case '>':
1665                 esc = esc_gt
1666             default:
1667                 continue
1668         }
1669         w.Write(s[last:i])

```

```

1670             w.Write(esc)
1671             last = i + 1
1672         }
1673         w.Write(s[last:])
1674     }
1675
1676     // procInstEncoding parses the `encoding="..."` or `encoding
1677     // value out of the provided string, returning "" if not fou
1678     func procInstEncoding(s string) string {
1679         // TODO: this parsing is somewhat lame and not exact
1680         // It works for all actual cases, though.
1681         idx := strings.Index(s, "encoding=")
1682         if idx == -1 {
1683             return ""
1684         }
1685         v := s[idx+len("encoding="):]
1686         if v == "" {
1687             return ""
1688         }
1689         if v[0] != '\\' && v[0] != '"' {
1690             return ""
1691         }
1692         idx = strings.IndexRune(v[1:], rune(v[0]))
1693         if idx == -1 {
1694             return ""
1695         }
1696         return v[1 : idx+1]
1697     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/errors/errors.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package errors implements functions to manipulate errors.
6 package errors
7
8 // New returns an error that formats as the given text.
9 func New(text string) error {
10     return &errorString{text}
11 }
12
13 // errorString is a trivial implementation of error.
14 type errorString struct {
15     s string
16 }
17
18 func (e *errorString) Error() string {
19     return e.s
20 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/expvar/expvar.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package expvar provides a standardized interface to publi
6 // as operation counters in servers. It exposes these variab
7 // /debug/vars in JSON format.
8 //
9 // Operations to set or modify these public variables are at
10 //
11 // In addition to adding the HTTP handler, this package regi
12 // following variables:
13 //
14 //     cmdline    os.Args
15 //     memstats   runtime.Memstats
16 //
17 // The package is sometimes only imported for the side effec
18 // registering its HTTP handler and the above variables. To
19 // this way, link this package into your program:
20 //     import _ "expvar"
21 //
22 package expvar
23
24 import (
25     "bytes"
26     "encoding/json"
27     "fmt"
28     "log"
29     "net/http"
30     "os"
31     "runtime"
32     "strconv"
33     "sync"
34 )
35
36 // Var is an abstract type for all exported variables.
37 type Var interface {
38     String() string
39 }
40
41 // Int is a 64-bit integer variable that satisfies the Var i
42 type Int struct {
43     i int64
44     mu sync.RWMutex
```

```

45 }
46
47 func (v *Int) String() string {
48     v.mu.RLock()
49     defer v.mu.RUnlock()
50     return strconv.FormatInt(v.i, 10)
51 }
52
53 func (v *Int) Add(delta int64) {
54     v.mu.Lock()
55     defer v.mu.Unlock()
56     v.i += delta
57 }
58
59 func (v *Int) Set(value int64) {
60     v.mu.Lock()
61     defer v.mu.Unlock()
62     v.i = value
63 }
64
65 // Float is a 64-bit float variable that satisfies the Var i
66 type Float struct {
67     f float64
68     mu sync.RWMutex
69 }
70
71 func (v *Float) String() string {
72     v.mu.RLock()
73     defer v.mu.RUnlock()
74     return strconv.FormatFloat(v.f, 'g', -1, 64)
75 }
76
77 // Add adds delta to v.
78 func (v *Float) Add(delta float64) {
79     v.mu.Lock()
80     defer v.mu.Unlock()
81     v.f += delta
82 }
83
84 // Set sets v to value.
85 func (v *Float) Set(value float64) {
86     v.mu.Lock()
87     defer v.mu.Unlock()
88     v.f = value
89 }
90
91 // Map is a string-to-Var map variable that satisfies the Va
92 type Map struct {
93     m map[string]Var
94     mu sync.RWMutex

```

```

95 }
96
97 // KeyValue represents a single entry in a Map.
98 type KeyValue struct {
99     Key    string
100    Value  Var
101 }
102
103 func (v *Map) String() string {
104     v.mu.RLock()
105     defer v.mu.RUnlock()
106     var b bytes.Buffer
107     fmt.Fprintf(&b, "{")
108     first := true
109     for key, val := range v.m {
110         if !first {
111             fmt.Fprintf(&b, ", ")
112         }
113         fmt.Fprintf(&b, "\"%s\": %v", key, val)
114         first = false
115     }
116     fmt.Fprintf(&b, "}")
117     return b.String()
118 }
119
120 func (v *Map) Init() *Map {
121     v.m = make(map[string]Var)
122     return v
123 }
124
125 func (v *Map) Get(key string) Var {
126     v.mu.RLock()
127     defer v.mu.RUnlock()
128     return v.m[key]
129 }
130
131 func (v *Map) Set(key string, av Var) {
132     v.mu.Lock()
133     defer v.mu.Unlock()
134     v.m[key] = av
135 }
136
137 func (v *Map) Add(key string, delta int64) {
138     v.mu.RLock()
139     av, ok := v.m[key]
140     v.mu.RUnlock()
141     if !ok {
142         // check again under the write lock
143         v.mu.Lock()

```

```

144         if _, ok = v.m[key]; !ok {
145             av = new(Int)
146             v.m[key] = av
147         }
148         v.mu.Unlock()
149     }
150
151     // Add to Int; ignore otherwise.
152     if iv, ok := av.(*Int); ok {
153         iv.Add(delta)
154     }
155 }
156
157 // AddFloat adds delta to the *Float value stored under the
158 func (v *Map) AddFloat(key string, delta float64) {
159     v.mu.RLock()
160     av, ok := v.m[key]
161     v.mu.RUnlock()
162     if !ok {
163         // check again under the write lock
164         v.mu.Lock()
165         if _, ok = v.m[key]; !ok {
166             av = new(Float)
167             v.m[key] = av
168         }
169         v.mu.Unlock()
170     }
171
172     // Add to Float; ignore otherwise.
173     if iv, ok := av.(*Float); ok {
174         iv.Add(delta)
175     }
176 }
177
178 // Do calls f for each entry in the map.
179 // The map is locked during the iteration,
180 // but existing entries may be concurrently updated.
181 func (v *Map) Do(f func(KeyValue)) {
182     v.mu.RLock()
183     defer v.mu.RUnlock()
184     for k, v := range v.m {
185         f(KeyValue{k, v})
186     }
187 }
188
189 // String is a string variable, and satisfies the Var interf
190 type String struct {
191     s string
192     mu sync.RWMutex

```

```

193 }
194
195 func (v *String) String() string {
196     v.mu.RLock()
197     defer v.mu.RUnlock()
198     return strconv.Quote(v.s)
199 }
200
201 func (v *String) Set(value string) {
202     v.mu.Lock()
203     defer v.mu.Unlock()
204     v.s = value
205 }
206
207 // Func implements Var by calling the function
208 // and formatting the returned value using JSON.
209 type Func func() interface{}
210
211 func (f Func) String() string {
212     v, _ := json.Marshal(f())
213     return string(v)
214 }
215
216 // All published variables.
217 var (
218     mutex sync.RWMutex
219     vars  map[string]Var = make(map[string]Var)
220 )
221
222 // Publish declares a named exported variable. This should be
223 // package's init function when it creates its Vars. If the
224 // registered then this will log.Panic.
225 func Publish(name string, v Var) {
226     mutex.Lock()
227     defer mutex.Unlock()
228     if _, existing := vars[name]; existing {
229         log.Panicln("Reuse of exported var name:", name)
230     }
231     vars[name] = v
232 }
233
234 // Get retrieves a named exported variable.
235 func Get(name string) Var {
236     mutex.RLock()
237     defer mutex.RUnlock()
238     return vars[name]
239 }
240
241 // Convenience functions for creating new exported variables
242

```

```

243 func NewInt(name string) *Int {
244     v := new(Int)
245     Publish(name, v)
246     return v
247 }
248
249 func NewFloat(name string) *Float {
250     v := new(Float)
251     Publish(name, v)
252     return v
253 }
254
255 func NewMap(name string) *Map {
256     v := new(Map).Init()
257     Publish(name, v)
258     return v
259 }
260
261 func NewString(name string) *String {
262     v := new(String)
263     Publish(name, v)
264     return v
265 }
266
267 // Do calls f for each exported variable.
268 // The global variable map is locked during the iteration,
269 // but existing entries may be concurrently updated.
270 func Do(f func(KeyValue)) {
271     mutex.RLock()
272     defer mutex.RUnlock()
273     for k, v := range vars {
274         f(KeyValue{k, v})
275     }
276 }
277
278 func expvarHandler(w http.ResponseWriter, r *http.Request) {
279     w.Header().Set("Content-Type", "application/json; ch
280     fmt.Fprintf(w, "{\n")
281     first := true
282     Do(func(kv KeyValue) {
283         if !first {
284             fmt.Fprintf(w, ",\n")
285         }
286         first = false
287         fmt.Fprintf(w, "%q: %s", kv.Key, kv.Value)
288     })
289     fmt.Fprintf(w, "\n}\n")
290 }
291

```

```
292 func cmdline() interface{} {
293     return os.Args
294 }
295
296 func memstats() interface{} {
297     stats := new(runtime.MemStats)
298     runtime.ReadMemStats(stats)
299     return *stats
300 }
301
302 func init() {
303     http.HandleFunc("/debug/vars", expvarHandler)
304     Publish("cmdline", Func(cmdline))
305     Publish("memstats", Func(memstats))
306 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/flag/flag.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6     Package flag implements command-line flag parsing.
7
8     Usage:
9
10    Define flags using flag.String(), Bool(), Int(), etc
11
12    This declares an integer flag, -flagname, stored in
13        import "flag"
14        var ip = flag.Int("flagname", 1234, "help me
15    If you like, you can bind the flag to a variable using
16        var flagvar int
17        func init() {
18            flag.IntVar(&flagvar, "flagname", 12
19        }
20    Or you can create custom flags that satisfy the Value
21    interface (and couple them to flag parsing by using
22        flag.Var(&flagVal, "name", "help message for
23    For such flags, the default value is just the initial
24
25    After all flags are defined, call
26        flag.Parse()
27    to parse the command line into the defined flags.
28
29    Flags may then be used directly. If you're using the Value
30    interface they are all pointers; if you bind to variables, the
31        fmt.Println("ip has value ", *ip)
32        fmt.Println("flagvar has value ", flagvar)
33
34    After parsing, the arguments after the flag are available as a
35    slice flag.Args() or individually as flag.Arg(i).
36    The arguments are indexed from 0 up to flag.NArg().
37
38    Command line flag syntax:
39        -flag
40        -flag=x
41        -flag x // non-boolean flags only
42    One or two minus signs may be used; they are equivalent.
43    The last form is not permitted for boolean flags because of the
44    meaning of the command
```

```

45         cmd -x *
46     will change if there is a file called 0, false, etc.
47     use the -flag=false form to turn off a boolean flag.
48
49     Flag parsing stops just before the first non-flag ar
50     ("- " is a non-flag argument) or after the terminator
51
52     Integer flags accept 1234, 0664, 0x1234 and may be n
53     Boolean flags may be 1, 0, t, f, true, false, TRUE,
54     Duration flags accept any input valid for time.Parse
55
56     The default set of command-line flags is controlled
57     top-level functions. The FlagSet type allows one to
58     independent sets of flags, such as to implement subc
59     in a command-line interface. The methods of FlagSet
60     analogous to the top-level functions for the command
61     flag set.
62 */
63 package flag
64
65 import (
66     "errors"
67     "fmt"
68     "io"
69     "os"
70     "sort"
71     "strconv"
72     "time"
73 )
74
75 // ErrHelp is the error returned if the flag -help is invoke
76 var ErrHelp = errors.New("flag: help requested")
77
78 // -- bool Value
79 type boolValue bool
80
81 func newBoolValue(val bool, p *bool) *boolValue {
82     *p = val
83     return (*boolValue)(p)
84 }
85
86 func (b *boolValue) Set(s string) error {
87     v, err := strconv.ParseBool(s)
88     *b = boolValue(v)
89     return err
90 }
91
92 func (b *boolValue) String() string { return fmt.Sprintf("%v", b) }
93
94 // -- int Value

```

```

95 type intValue int
96
97 func newIntValue(val int, p *int) *intValue {
98     *p = val
99     return (*intValue)(p)
100 }
101
102 func (i *intValue) Set(s string) error {
103     v, err := strconv.ParseInt(s, 0, 64)
104     *i = intValue(v)
105     return err
106 }
107
108 func (i *intValue) String() string { return fmt.Sprintf("%v"
109
110 // -- int64 Value
111 type int64Value int64
112
113 func newInt64Value(val int64, p *int64) *int64Value {
114     *p = val
115     return (*int64Value)(p)
116 }
117
118 func (i *int64Value) Set(s string) error {
119     v, err := strconv.ParseInt(s, 0, 64)
120     *i = int64Value(v)
121     return err
122 }
123
124 func (i *int64Value) String() string { return fmt.Sprintf("%
125
126 // -- uint Value
127 type uintValue uint
128
129 func newUIntValue(val uint, p *uint) *uintValue {
130     *p = val
131     return (*uintValue)(p)
132 }
133
134 func (i *uintValue) Set(s string) error {
135     v, err := strconv.ParseUint(s, 0, 64)
136     *i = uintValue(v)
137     return err
138 }
139
140 func (i *uintValue) String() string { return fmt.Sprintf("%v
141
142 // -- uint64 Value
143 type uint64Value uint64

```

```

144
145 func newUint64Value(val uint64, p *uint64) *uint64Value {
146     *p = val
147     return (*uint64Value)(p)
148 }
149
150 func (i *uint64Value) Set(s string) error {
151     v, err := strconv.ParseUint(s, 0, 64)
152     *i = uint64Value(v)
153     return err
154 }
155
156 func (i *uint64Value) String() string { return fmt.Sprintf("
157
158 // -- string Value
159 type stringValue string
160
161 func newStringValue(val string, p *string) *stringValue {
162     *p = val
163     return (*stringValue)(p)
164 }
165
166 func (s *stringValue) Set(val string) error {
167     *s = stringValue(val)
168     return nil
169 }
170
171 func (s *stringValue) String() string { return fmt.Sprintf("
172
173 // -- float64 Value
174 type float64Value float64
175
176 func newFloat64Value(val float64, p *float64) *float64Value
177     *p = val
178     return (*float64Value)(p)
179 }
180
181 func (f *float64Value) Set(s string) error {
182     v, err := strconv.ParseFloat(s, 64)
183     *f = float64Value(v)
184     return err
185 }
186
187 func (f *float64Value) String() string { return fmt.Sprintf(
188
189 // -- time.Duration Value
190 type durationValue time.Duration
191
192 func newDurationValue(val time.Duration, p *time.Duration) *

```

```

193         *p = val
194         return (*durationValue)(p)
195     }
196
197     func (d *durationValue) Set(s string) error {
198         v, err := time.ParseDuration(s)
199         *d = durationValue(v)
200         return err
201     }
202
203     func (d *durationValue) String() string { return (*time.Dura
204
205     // Value is the interface to the dynamic value stored in a f
206     // (The default value is represented as a string.)
207     type Value interface {
208         String() string
209         Set(string) error
210     }
211
212     // ErrorHandler defines how to handle flag parsing errors.
213     type ErrorHandler int
214
215     const (
216         ContinueOnError ErrorHandler = iota
217         ExitOnError
218         PanicOnError
219     )
220
221     // A FlagSet represents a set of defined flags.
222     type FlagSet struct {
223         // Usage is the function called when an error occurs
224         // The field is a function (not a method) that may b
225         // a custom error handler.
226         Usage func()
227
228         name          string
229         parsed        bool
230         actual        map[string]*Flag
231         formal        map[string]*Flag
232         args          []string // arguments after flags
233         exitOnError   bool     // does the program exit if t
234         errorHandler ErrorHandler
235         output        io.Writer // nil means stderr; use out
236     }
237
238     // A Flag represents the state of a flag.
239     type Flag struct {
240         Name     string // name as it appears on command lin
241         Usage    string // help message
242         Value    Value  // value as set

```

```

243         DefValue string // default value (as text); for usag
244     }
245
246 // sortFlags returns the flags as a slice in lexicographical
247 func sortFlags(flags map[string]*Flag) []*Flag {
248     list := make(sort.StringSlice, len(flags))
249     i := 0
250     for _, f := range flags {
251         list[i] = f.Name
252         i++
253     }
254     list.Sort()
255     result := make([]*Flag, len(list))
256     for i, name := range list {
257         result[i] = flags[name]
258     }
259     return result
260 }
261
262 func (f *FlagSet) out() io.Writer {
263     if f.output == nil {
264         return os.Stderr
265     }
266     return f.output
267 }
268
269 // SetOutput sets the destination for usage and error messag
270 // If output is nil, os.Stderr is used.
271 func (f *FlagSet) SetOutput(output io.Writer) {
272     f.output = output
273 }
274
275 // VisitAll visits the flags in lexicographical order, calli
276 // It visits all flags, even those not set.
277 func (f *FlagSet) VisitAll(fn func(*Flag)) {
278     for _, flag := range sortFlags(f.formal) {
279         fn(flag)
280     }
281 }
282
283 // VisitAll visits the command-line flags in lexicographical
284 // fn for each. It visits all flags, even those not set.
285 func VisitAll(fn func(*Flag)) {
286     commandLine.VisitAll(fn)
287 }
288
289 // Visit visits the flags in lexicographical order, calling
290 // It visits only those flags that have been set.
291 func (f *FlagSet) Visit(fn func(*Flag)) {

```

```

292         for _, flag := range sortFlags(f.actual) {
293             fn(flag)
294         }
295     }
296
297     // Visit visits the command-line flags in lexicographical or
298     // for each. It visits only those flags that have been set.
299     func Visit(fn func(*Flag)) {
300         commandLine.Visit(fn)
301     }
302
303     // Lookup returns the Flag structure of the named flag, retu
304     func (f *FlagSet) Lookup(name string) *Flag {
305         return f.formal[name]
306     }
307
308     // Lookup returns the Flag structure of the named command-li
309     // returning nil if none exists.
310     func Lookup(name string) *Flag {
311         return commandLine.formal[name]
312     }
313
314     // Set sets the value of the named flag.
315     func (f *FlagSet) Set(name, value string) error {
316         flag, ok := f.formal[name]
317         if !ok {
318             return fmt.Errorf("no such flag -%v", name)
319         }
320         err := flag.Value.Set(value)
321         if err != nil {
322             return err
323         }
324         if f.actual == nil {
325             f.actual = make(map[string]*Flag)
326         }
327         f.actual[name] = flag
328         return nil
329     }
330
331     // Set sets the value of the named command-line flag.
332     func Set(name, value string) error {
333         return commandLine.Set(name, value)
334     }
335
336     // PrintDefaults prints, to standard error unless configured
337     // otherwise, the default values of all defined flags in the
338     func (f *FlagSet) PrintDefaults() {
339         f.VisitAll(func(flag *Flag) {
340             format := "  -%s=%s: %s\n"

```

```

341         if _, ok := flag.Value.(*stringValue); ok {
342             // put quotes on the value
343             format = "  -%s=%q: %s\n"
344         }
345         fmt.Fprintf(f.out(), format, flag.Name, flag
346     })
347 }
348
349 // PrintDefaults prints to standard error the default values
350 func PrintDefaults() {
351     commandLine.PrintDefaults()
352 }
353
354 // defaultUsage is the default function to print a usage mes
355 func defaultUsage(f *FlagSet) {
356     fmt.Fprintf(f.out(), "Usage of %s:\n", f.name)
357     f.PrintDefaults()
358 }
359
360 // NOTE: Usage is not just defaultUsage(commandLine)
361 // because it serves (via godoc flag Usage) as the example
362 // for how to write your own usage function.
363
364 // Usage prints to standard error a usage message documentin
365 // The function is a variable that may be changed to point t
366 var Usage = func() {
367     fmt.Fprintf(os.Stderr, "Usage of %s:\n", os.Args[0])
368     PrintDefaults()
369 }
370
371 // NFlag returns the number of flags that have been set.
372 func (f *FlagSet) NFlag() int { return len(f.actual) }
373
374 // NFlag returns the number of command-line flags that have
375 func NFlag() int { return len(commandLine.actual) }
376
377 // Arg returns the i'th argument. Arg(0) is the first remai
378 // after flags have been processed.
379 func (f *FlagSet) Arg(i int) string {
380     if i < 0 || i >= len(f.args) {
381         return ""
382     }
383     return f.args[i]
384 }
385
386 // Arg returns the i'th command-line argument. Arg(0) is th
387 // after flags have been processed.
388 func Arg(i int) string {
389     return commandLine.Arg(i)
390 }

```

```

391
392 // NArg is the number of arguments remaining after flags hav
393 func (f *FlagSet) NArg() int { return len(f.args) }
394
395 // NArg is the number of arguments remaining after flags hav
396 func NArg() int { return len(commandLine.args) }
397
398 // Args returns the non-flag arguments.
399 func (f *FlagSet) Args() []string { return f.args }
400
401 // Args returns the non-flag command-line arguments.
402 func Args() []string { return commandLine.args }
403
404 // BoolVar defines a bool flag with specified name, default
405 // The argument p points to a bool variable in which to stor
406 func (f *FlagSet) BoolVar(p *bool, name string, value bool,
407     f.Var(newBoolValue(value, p), name, usage)
408 }
409
410 // BoolVar defines a bool flag with specified name, default
411 // The argument p points to a bool variable in which to stor
412 func BoolVar(p *bool, name string, value bool, usage string)
413     commandLine.Var(newBoolValue(value, p), name, usage)
414 }
415
416 // Bool defines a bool flag with specified name, default val
417 // The return value is the address of a bool variable that s
418 func (f *FlagSet) Bool(name string, value bool, usage string
419     p := new(bool)
420     f.BoolVar(p, name, value, usage)
421     return p
422 }
423
424 // Bool defines a bool flag with specified name, default val
425 // The return value is the address of a bool variable that s
426 func Bool(name string, value bool, usage string) *bool {
427     return commandLine.Bool(name, value, usage)
428 }
429
430 // IntVar defines an int flag with specified name, default v
431 // The argument p points to an int variable in which to stor
432 func (f *FlagSet) IntVar(p *int, name string, value int, usa
433     f.Var(newIntValue(value, p), name, usage)
434 }
435
436 // IntVar defines an int flag with specified name, default v
437 // The argument p points to an int variable in which to stor
438 func IntVar(p *int, name string, value int, usage string) {
439     commandLine.Var(newIntValue(value, p), name, usage)

```

```

440 }
441
442 // Int defines an int flag with specified name, default valu
443 // The return value is the address of an int variable that s
444 func (f *FlagSet) Int(name string, value int, usage string)
445     p := new(int)
446     f.IntVar(p, name, value, usage)
447     return p
448 }
449
450 // Int defines an int flag with specified name, default valu
451 // The return value is the address of an int variable that s
452 func Int(name string, value int, usage string) *int {
453     return CommandLine.Int(name, value, usage)
454 }
455
456 // Int64Var defines an int64 flag with specified name, defau
457 // The argument p points to an int64 variable in which to st
458 func (f *FlagSet) Int64Var(p *int64, name string, value int6
459     f.Var(newInt64Value(value, p), name, usage)
460 }
461
462 // Int64Var defines an int64 flag with specified name, defau
463 // The argument p points to an int64 variable in which to st
464 func Int64Var(p *int64, name string, value int64, usage stri
465     CommandLine.Var(newInt64Value(value, p), name, usage)
466 }
467
468 // Int64 defines an int64 flag with specified name, default
469 // The return value is the address of an int64 variable that
470 func (f *FlagSet) Int64(name string, value int64, usage stri
471     p := new(int64)
472     f.Int64Var(p, name, value, usage)
473     return p
474 }
475
476 // Int64 defines an int64 flag with specified name, default
477 // The return value is the address of an int64 variable that
478 func Int64(name string, value int64, usage string) *int64 {
479     return CommandLine.Int64(name, value, usage)
480 }
481
482 // UintVar defines a uint flag with specified name, default
483 // The argument p points to a uint variable in which to stor
484 func (f *FlagSet) UintVar(p *uint, name string, value uint,
485     f.Var(newUintValue(value, p), name, usage)
486 }
487
488 // UintVar defines a uint flag with specified name, default

```

```

489 // The argument p points to a uint variable in which to sto
490 func UintVar(p *uint, name string, value uint, usage string)
491     CommandLine.Var(newUintValue(value, p), name, usage)
492 }
493
494 // Uint defines a uint flag with specified name, default val
495 // The return value is the address of a uint variable that
496 func (f *FlagSet) Uint(name string, value uint, usage string)
497     p := new(uint)
498     f.UintVar(p, name, value, usage)
499     return p
500 }
501
502 // Uint defines a uint flag with specified name, default val
503 // The return value is the address of a uint variable that
504 func Uint(name string, value uint, usage string) *uint {
505     return CommandLine.Uint(name, value, usage)
506 }
507
508 // Uint64Var defines a uint64 flag with specified name, defa
509 // The argument p points to a uint64 variable in which to st
510 func (f *FlagSet) Uint64Var(p *uint64, name string, value ui
511     f.Var(newUint64Value(value, p), name, usage)
512 }
513
514 // Uint64Var defines a uint64 flag with specified name, defa
515 // The argument p points to a uint64 variable in which to st
516 func Uint64Var(p *uint64, name string, value uint64, usage s
517     CommandLine.Var(newUint64Value(value, p), name, usag
518 }
519
520 // Uint64 defines a uint64 flag with specified name, default
521 // The return value is the address of a uint64 variable that
522 func (f *FlagSet) Uint64(name string, value uint64, usage st
523     p := new(uint64)
524     f.Uint64Var(p, name, value, usage)
525     return p
526 }
527
528 // Uint64 defines a uint64 flag with specified name, default
529 // The return value is the address of a uint64 variable that
530 func Uint64(name string, value uint64, usage string) *uint64
531     return CommandLine.Uint64(name, value, usage)
532 }
533
534 // StringVar defines a string flag with specified name, defa
535 // The argument p points to a string variable in which to st
536 func (f *FlagSet) StringVar(p *string, name string, value st
537     f.Var(newStringValue(value, p), name, usage)
538 }

```

```

539
540 // StringVar defines a string flag with specified name, defa
541 // The argument p points to a string variable in which to st
542 func StringVar(p *string, name string, value string, usage s
543     CommandLine.Var(newStringValue(value, p), name, usag
544 }
545
546 // String defines a string flag with specified name, default
547 // The return value is the address of a string variable that
548 func (f *FlagSet) String(name string, value string, usage st
549     p := new(string)
550     f.StringVar(p, name, value, usage)
551     return p
552 }
553
554 // String defines a string flag with specified name, default
555 // The return value is the address of a string variable that
556 func String(name string, value string, usage string) *string
557     return CommandLine.String(name, value, usage)
558 }
559
560 // Float64Var defines a float64 flag with specified name, de
561 // The argument p points to a float64 variable in which to s
562 func (f *FlagSet) Float64Var(p *float64, name string, value
563     f.Var(newFloat64Value(value, p), name, usage)
564 }
565
566 // Float64Var defines a float64 flag with specified name, de
567 // The argument p points to a float64 variable in which to s
568 func Float64Var(p *float64, name string, value float64, usag
569     CommandLine.Var(newFloat64Value(value, p), name, usa
570 }
571
572 // Float64 defines a float64 flag with specified name, defau
573 // The return value is the address of a float64 variable tha
574 func (f *FlagSet) Float64(name string, value float64, usage
575     p := new(float64)
576     f.Float64Var(p, name, value, usage)
577     return p
578 }
579
580 // Float64 defines a float64 flag with specified name, defau
581 // The return value is the address of a float64 variable tha
582 func Float64(name string, value float64, usage string) *floo
583     return CommandLine.Float64(name, value, usage)
584 }
585
586 // DurationVar defines a time.Duration flag with specified n
587 // The argument p points to a time.Duration variable in whic

```

```

588 func (f *FlagSet) DurationVar(p *time.Duration, name string,
589     f.Var(newDurationValue(value, p), name, usage)
590 }
591
592 // DurationVar defines a time.Duration flag with specified n
593 // The argument p points to a time.Duration variable in which
594 func DurationVar(p *time.Duration, name string, value time.D
595     commandLine.Var(newDurationValue(value, p), name, us
596 }
597
598 // Duration defines a time.Duration flag with specified name
599 // The return value is the address of a time.Duration variab
600 func (f *FlagSet) Duration(name string, value time.Duration,
601     p := new(time.Duration)
602     f.DurationVar(p, name, value, usage)
603     return p
604 }
605
606 // Duration defines a time.Duration flag with specified name
607 // The return value is the address of a time.Duration variab
608 func Duration(name string, value time.Duration, usage string
609     return commandLine.Duration(name, value, usage)
610 }
611
612 // Var defines a flag with the specified name and usage stri
613 // value of the flag are represented by the first argument,
614 // typically holds a user-defined implementation of Value. F
615 // caller could create a flag that turns a comma-separated s
616 // of strings by giving the slice the methods of Value; in p
617 // decompose the comma-separated string into the slice.
618 func (f *FlagSet) Var(value Value, name string, usage string
619     // Remember the default value as a string; it won't
620     flag := &Flag{name, usage, value, value.String()}
621     _, alreadythere := f.formal[name]
622     if alreadythere {
623         fmt.Fprintf(f.out(), "%s flag redefined: %s\
624             panic("flag redefinition") // Happens only i
625     }
626     if f.formal == nil {
627         f.formal = make(map[string]*Flag)
628     }
629     f.formal[name] = flag
630 }
631
632 // Var defines a flag with the specified name and usage stri
633 // value of the flag are represented by the first argument,
634 // typically holds a user-defined implementation of Value. F
635 // caller could create a flag that turns a comma-separated s
636 // of strings by giving the slice the methods of Value; in p

```

```

637 // decompose the comma-separated string into the slice.
638 func Var(value Value, name string, usage string) {
639     commandLine.Var(value, name, usage)
640 }
641
642 // failf prints to standard error a formatted error and usage
643 // returns the error.
644 func (f *FlagSet) failf(format string, a ...interface{}) error {
645     err := fmt.Errorf(format, a...)
646     fmt.Fprintln(f.out(), err)
647     f.usage()
648     return err
649 }
650
651 // usage calls the Usage method for the flag set, or the usage
652 // the flag set is commandLine.
653 func (f *FlagSet) usage() {
654     if f == commandLine {
655         Usage()
656     } else if f.Usage == nil {
657         defaultUsage(f)
658     } else {
659         f.Usage()
660     }
661 }
662
663 // parseOne parses one flag. It returns whether a flag was seen
664 func (f *FlagSet) parseOne() (bool, error) {
665     if len(f.args) == 0 {
666         return false, nil
667     }
668     s := f.args[0]
669     if len(s) == 0 || s[0] != '-' || len(s) == 1 {
670         return false, nil
671     }
672     num_minuses := 1
673     if s[1] == '-' {
674         num_minuses++
675         if len(s) == 2 { // "--" terminates the flag
676             f.args = f.args[1:]
677             return false, nil
678         }
679     }
680     name := s[num_minuses:]
681     if len(name) == 0 || name[0] == '-' || name[0] == '=' {
682         return false, f.failf("bad flag syntax: %s",
683             name)
684     }
685     // it's a flag. does it have an argument?
686     f.args = f.args[1:]

```

```

687     has_value := false
688     value := ""
689     for i := 1; i < len(name); i++ { // equals cannot be
690         if name[i] == '=' {
691             value = name[i+1:]
692             has_value = true
693             name = name[0:i]
694             break
695         }
696     }
697     m := f.formal
698     flag, alreadythere := m[name] // BUG
699     if !alreadythere {
700         if name == "help" || name == "h" { // special
701             f.usage()
702             return false, ErrHelp
703         }
704         return false, f.failf("flag provided but not
705     }
706     if fv, ok := flag.Value.(*boolValue); ok { // special
707         if has_value {
708             if err := fv.Set(value); err != nil
709                 f.failf("invalid boolean val
710         }
711         } else {
712             fv.Set("true")
713         }
714     } else {
715         // It must have a value, which might be the
716         if !has_value && len(f.args) > 0 {
717             // value is the next arg
718             has_value = true
719             value, f.args = f.args[0], f.args[1:
720         }
721         if !has_value {
722             return false, f.failf("flag needs an
723         }
724         if err := flag.Value.Set(value); err != nil
725             return false, f.failf("invalid value
726         }
727     }
728     if f.actual == nil {
729         f.actual = make(map[string]*Flag)
730     }
731     f.actual[name] = flag
732     return true, nil
733 }
734
735 // Parse parses flag definitions from the argument list, whi

```

```

736 // include the command name. Must be called after all flags
737 // are defined and before flags are accessed by the program.
738 // The return value will be ErrHelp if -help was set but not
739 func (f *FlagSet) Parse(arguments []string) error {
740     f.parsed = true
741     f.args = arguments
742     for {
743         seen, err := f.parseOne()
744         if seen {
745             continue
746         }
747         if err == nil {
748             break
749         }
750         switch f.errorHandling {
751         case ContinueOnError:
752             return err
753         case ExitOnError:
754             os.Exit(2)
755         case PanicOnError:
756             panic(err)
757         }
758     }
759     return nil
760 }
761
762 // Parsed reports whether f.Parse has been called.
763 func (f *FlagSet) Parsed() bool {
764     return f.parsed
765 }
766
767 // Parse parses the command-line flags from os.Args[1:]. Mu
768 // after all flags are defined and before flags are accessed
769 func Parse() {
770     // Ignore errors; commandLine is set for ExitOnError
771     commandLine.Parse(os.Args[1:])
772 }
773
774 // Parsed returns true if the command-line flags have been p
775 func Parsed() bool {
776     return commandLine.Parsed()
777 }
778
779 // The default set of command-line flags, parsed from os.Arg
780 var commandLine = NewFlagSet(os.Args[0], ExitOnError)
781
782 // NewFlagSet returns a new, empty flag set with the specifi
783 // error handling property.
784 func NewFlagSet(name string, errorHandling ErrorHandling) *F

```

```
785         f := &FlagSet{
786             name:         name,
787             errorHandling: errorHandling,
788         }
789         return f
790     }
791
792     // Init sets the name and error handling property for a flag
793     // By default, the zero FlagSet uses an empty name and the
794     // ContinueOnError error handling policy.
795     func (f *FlagSet) Init(name string, errorHandling ErrorHandl
796         f.name = name
797         f.errorHandling = errorHandling
798     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/fmt/doc.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6     Package fmt implements formatted I/O with functions
7     to C's printf and scanf. The format 'verbs' are der
8     are simpler.
9
10
11     Printing
12
13     The verbs:
14
15     General:
16         %v          the value in a default format.
17                    when printing structs, the plus flag
18         %#v        a Go-syntax representation of the va
19         %T          a Go-syntax representation of the ty
20         %%          a literal percent sign; consumes no
21
22     Boolean:
23         %t          the word true or false
24
25     Integer:
26         %b          base 2
27         %c          the character represented by the cor
28         %d          base 10
29         %o          base 8
30         %q          a single-quoted character literal sa
31         %x          base 16, with lower-case letters for
32         %X          base 16, with upper-case letters for
33         %U          Unicode format: U+1234; same as "U+%
34
35     Floating-point and complex constituents:
36         %b          decimalless scientific notation with
37                    in the manner of strconv.FormatFloat
38                    e.g. -123456p-78
39         %e          scientific notation, e.g. -1234.456e
40         %E          scientific notation, e.g. -1234.456E
41         %f          decimal point but no exponent, e.g.
42         %g          whichever of %e or %f produces more
43         %G          whichever of %E or %f produces more
44
45     String and slice of bytes:
46         %s          the uninterpreted bytes of the strin
47         %q          a double-quoted string safely escape
```

45 %x base 16, lower-case, two characters
46 %X base 16, upper-case, two characters
47 Pointer:
48 %p base 16 notation, with leading 0x
49

50 There is no 'u' flag. Integers are printed unsigned
51 Similarly, there is no need to specify the size of t
52

53 The width and precision control formatting and are i
54 code points. (This differs from C's printf where th
55 of bytes.) Either or both of the flags may be replac
56 character '*', causing their values to be obtained f
57 operand, which must be of type int.
58

59 For numeric values, width sets the width of the fiel
60 sets the number of places after the decimal, if appr
61 example, the format %6.2f prints 123.45.
62

63 For strings, width is the minimum number of characte
64 padding with spaces if necessary, and precision is t
65 number of characters to output, truncating if necess
66

67 Other flags:

68 + always print a sign for numeric valu
69 guarantee ASCII-only output for %q (
70 - pad with spaces on the right rather
71 # alternate format: add leading 0 for
72 0X for hex (%#X); suppress 0x for %p
73 print a raw (backquoted) string if p
74 write e.g. U+0078 'x' if the charact
75 ' ' (space) leave a space for elided sig
76 put spaces between bytes printing st
77 0 pad with leading zeros rather than s
78

79 For each Printf-like function, there is also a Print
80 that takes no format and is equivalent to saying %v
81 operand. Another variant Println inserts blanks bet
82 operands and appends a newline.
83

84 Regardless of the verb, if an operand is an interfac
85 the internal concrete value is used, not the interfa
86 Thus:

```
87                    var i interface{} = 23  
88                    fmt.Printf("%v\n", i)
```

89 will print 23.
90

91 If an operand implements interface Formatter, that i
92 can be used for fine control of formatting.
93

94 If the format (which is implicitly %v for Println et

95 for a string (%s %q %v %x %X), the following two rul
96
97 1. If an operand implements the error interface, the
98 will be used to convert the object to a string, whic
99 be formatted as required by the verb (if any).
100
101 2. If an operand implements method String() string,
102 will be used to convert the object to a string, whic
103 be formatted as required by the verb (if any).
104
105 To avoid recursion in cases such as
106 type X string
107 func (x X) String() string { return Sprintf(
108 convert the value before recurring:
109 func (x X) String() string { return Sprintf(
110
111 Format errors:
112
113 If an invalid argument is given for a verb, such as
114 a string to %d, the generated string will contain a
115 description of the problem, as in these examples:
116
117 Wrong type or unknown verb: %!verb(type=valu
118 Printf("%d", hi): %!d(strin
119 Too many arguments: %!(EXTRA type=value)
120 Printf("hi", "guys"): hi%!(EXTR
121 Too few arguments: %!verb(MISSING)
122 Printf("hi%d"): hi %!d(MI
123 Non-int for width or precision: %!(BADWIDTH)
124 Printf("%*s", 4.5, "hi"): %!(BADWID
125 Printf("%. *s", 4.5, "hi"): %!(BADPRE
126
127 All errors begin with the string "%!" followed somet
128 by a single character (the verb) and end with a pare
129 description.
130
131
132 Scanning
133
134 An analogous set of functions scans formatted text t
135 values. Scan, Scanf and Scanln read from os.Stdin;
136 Fscanf and Fscanln read from a specified io.Reader;
137 Sscanf and Sscanln read from an argument string. Sc
138 Fscanln and Sscanln stop scanning at a newline and r
139 the items be followed by one; Sscanf, Fscanf and Ssc
140 newlines in the input to match newlines in the forma
141 routines treat newlines as spaces.
142
143 Scanf, Fscanf, and Sscanf parse the arguments accord

```
144     format string, analogous to that of Printf. For exa
145 will scan an integer as a hexadecimal number, and %v
146 the default representation format for the value.
147
148     The formats behave analogously to those of Printf wi
149 following exceptions:
150
151         %p is not implemented
152         %T is not implemented
153         %e %E %f %F %g %G are all equivalent and sca
154         %s and %v on strings scan a space-delimited
155
156     The familiar base-setting prefixes 0 (octal) and 0x
157 (hexadecimal) are accepted when scanning integers wi
158 format or with the %v verb.
159
160     Width is interpreted in the input text (%5s means at
161 five runes of input will be read to scan a string) b
162 is no syntax for scanning with a precision (no %5.2f
163 %5f).
164
165     When scanning with a format, all non-empty runs of s
166 characters (except newline) are equivalent to a sing
167 space in both the format and the input. With that p
168 text in the format string must match the input text;
169 stops if it does not, with the return value of the f
170 indicating the number of arguments scanned.
171
172     In all the scanning functions, if an operand impleme
173 Scan (that is, it implements the Scanner interface)
174 method will be used to scan the text for that operan
175 if the number of arguments scanned is less than the
176 arguments provided, an error is returned.
177
178     All arguments to be scanned must be either pointers
179 types or implementations of the Scanner interface.
180
181     Note: Fscan etc. can read one character (rune) past
182 they return, which means that a loop calling a scan
183 may skip some of the input. This is usually a probl
184 when there is no space between input values. If the
185 provided to Fscan implements ReadRune, that method w
186 to read characters. If the reader also implements U
187 that method will be used to save the character and s
188 calls will not lose data. To attach ReadRune and Un
189 methods to a reader without that capability, use
190 bufio.NewReader.
191 */
192 package fmt
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/fmt/format.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package fmt
6
7 import (
8     "strconv"
9     "unicode/utf8"
10 )
11
12 const (
13     nByte = 65 // %b of an int64, plus a sign.
14
15     ldigits = "0123456789abcdef"
16     udigits = "0123456789ABCDEF"
17 )
18
19 const (
20     signed    = true
21     unsigned  = false
22 )
23
24 var padZeroBytes = make([]byte, nByte)
25 var padSpaceBytes = make([]byte, nByte)
26
27 var newline = []byte{'\n'}
28
29 func init() {
30     for i := 0; i < nByte; i++ {
31         padZeroBytes[i] = '0'
32         padSpaceBytes[i] = ' '
33     }
34 }
35
36 // A fmt is the raw formatter used by Printf etc.
37 // It prints into a buffer that must be set up separately.
38 type fmt struct {
39     intbuf [nByte]byte
40     buf    *buffer
41     // width, precision
42     wid  int
43     prec int
44     // flags
```

```

45     widPresent    bool
46     precPresent  bool
47     minus        bool
48     plus         bool
49     sharp        bool
50     space        bool
51     unicode      bool
52     uniQuote     bool // Use 'x' prefix for %U if printa
53     zero         bool
54 }
55
56 func (f *fmt) clearflags() {
57     f.wid = 0
58     f.widPresent = false
59     f.prec = 0
60     f.precPresent = false
61     f.minus = false
62     f.plus = false
63     f.sharp = false
64     f.space = false
65     f.unicode = false
66     f.uniQuote = false
67     f.zero = false
68 }
69
70 func (f *fmt) init(buf *buffer) {
71     f.buf = buf
72     f.clearflags()
73 }
74
75 // Compute left and right padding widths (only one will be n
76 func (f *fmt) computePadding(width int) (padding []byte, lef
77     left := !f.minus
78     w := f.wid
79     if w < 0 {
80         left = false
81         w = -w
82     }
83     w -= width
84     if w > 0 {
85         if left && f.zero {
86             return padZeroBytes, w, 0
87         }
88         if left {
89             return padSpaceBytes, w, 0
90         } else {
91             // can't be zero padding on the righ
92             return padSpaceBytes, 0, w
93         }
94     }

```

```

95         return
96     }
97
98     // Generate n bytes of padding.
99     func (f *fmt) writePadding(n int, padding []byte) {
100         for n > 0 {
101             m := n
102             if m > nByte {
103                 m = nByte
104             }
105             f.buf.Write(padding[0:m])
106             n -= m
107         }
108     }
109
110     // Append b to f.buf, padded on left (w > 0) or right (w < 0)
111     // clear flags afterwards.
112     func (f *fmt) pad(b []byte) {
113         var padding []byte
114         var left, right int
115         if f.widPresent && f.wid != 0 {
116             padding, left, right = f.computePadding(len(
117                 b))
118             if left > 0 {
119                 f.writePadding(left, padding)
120             }
121             f.buf.Write(b)
122             if right > 0 {
123                 f.writePadding(right, padding)
124             }
125         }
126
127     // append s to buf, padded on left (w > 0) or right (w < 0)
128     // clear flags afterwards.
129     func (f *fmt) padString(s string) {
130         var padding []byte
131         var left, right int
132         if f.widPresent && f.wid != 0 {
133             padding, left, right = f.computePadding(utf8
134                 len(s))
135             if left > 0 {
136                 f.writePadding(left, padding)
137             }
138             f.buf.WriteString(s)
139             if right > 0 {
140                 f.writePadding(right, padding)
141             }
142         }
143     }

```

```

144 func putint(buf []byte, base, val uint64, digits string) int
145     i := len(buf) - 1
146     for val >= base {
147         buf[i] = digits[val%base]
148         i--
149         val /= base
150     }
151     buf[i] = digits[val]
152     return i - 1
153 }
154
155 var (
156     trueBytes = []byte("true")
157     falseBytes = []byte("false")
158 )
159
160 // fmt_boolean formats a boolean.
161 func (f *fmt) fmt_boolean(v bool) {
162     if v {
163         f.pad(trueBytes)
164     } else {
165         f.pad(falseBytes)
166     }
167 }
168
169 // integer; interprets prec but not wid. Once formatted, re
170 // and then flags are cleared.
171 func (f *fmt) integer(a int64, base uint64, signedness bool,
172     // precision of 0 and value of 0 means "print nothin
173     if f.precPresent && f.prec == 0 && a == 0 {
174         return
175     }
176
177     var buf []byte = f.intbuf[0:]
178     negative := signedness == signed && a < 0
179     if negative {
180         a = -a
181     }
182
183     // two ways to ask for extra leading zero digits: %.
184     // apparently the first cancels the second.
185     prec := 0
186     if f.precPresent {
187         prec = f.prec
188         f.zero = false
189     } else if f.zero && f.widPresent && !f.minus && f.wi
190         prec = f.wid
191         if negative || f.plus || f.space {
192             prec-- // leave room for sign

```

```

193     }
194 }
195
196 // format a into buf, ending at buf[i]. (printing i
197 // a is made into unsigned ua. we could make things
198 // marginally faster by splitting the 32-bit case ou
199 // block but it's not worth the duplication, so ua h
200 i := len(f.intbuf)
201 ua := uint64(a)
202 for ua >= base {
203     i--
204     buf[i] = digits[ua%base]
205     ua /= base
206 }
207 i--
208 buf[i] = digits[ua]
209 for i > 0 && prec > nByte-i {
210     i--
211     buf[i] = '0'
212 }
213
214 // Various prefixes: 0x, -, etc.
215 if f.sharp {
216     switch base {
217     case 8:
218         if buf[i] != '0' {
219             i--
220             buf[i] = '0'
221         }
222     case 16:
223         i--
224         buf[i] = 'x' + digits[10] - 'a'
225         i--
226         buf[i] = '0'
227     }
228 }
229 if f.unicode {
230     i--
231     buf[i] = '+'
232     i--
233     buf[i] = 'U'
234 }
235
236 if negative {
237     i--
238     buf[i] = '-'
239 } else if f.plus {
240     i--
241     buf[i] = '+'
242 } else if f.space {

```

```

243         i--
244         buf[i] = ' '
245     }
246
247     // If we want a quoted char for %#U, move the data u
248     if f.unicode && f.uniQuote && a >= 0 && a <= utf8.Ma
249         runeWidth := utf8.RuneLen(rune(a))
250         width := 1 + 1 + runeWidth + 1 // space, quo
251         copy(buf[i-width:], buf[i:]) // guaranteed
252         i -= width
253         // Now put " 'x'" at the end.
254         j := len(buf) - width
255         buf[j] = ' '
256         j++
257         buf[j] = '\''
258         j++
259         utf8.EncodeRune(buf[j:], rune(a))
260         j += runeWidth
261         buf[j] = '\''
262     }
263
264     f.pad(buf[i:])
265 }
266
267 // truncate truncates the string to the specified precision,
268 func (f *fmt) truncate(s string) string {
269     if f.precPresent && f.prec < utf8.RuneCountInString(
270         n := f.prec
271         for i := range s {
272             if n == 0 {
273                 s = s[:i]
274                 break
275             }
276             n--
277         }
278     }
279     return s
280 }
281
282 // fmt_s formats a string.
283 func (f *fmt) fmt_s(s string) {
284     s = f.truncate(s)
285     f.padString(s)
286 }
287
288 // fmt_sx formats a string as a hexadecimal encoding of its
289 func (f *fmt) fmt_sx(s, digits string) {
290     // TODO: Avoid buffer by pre-padding.
291     var b []byte

```

```

292         for i := 0; i < len(s); i++ {
293             if i > 0 && f.space {
294                 b = append(b, ' ')
295             }
296             v := s[i]
297             b = append(b, digits[v>>4], digits[v&0xF])
298         }
299         f.pad(b)
300     }
301
302     // fmt_q formats a string as a double-quoted, escaped Go str
303     func (f *fmt) fmt_q(s string) {
304         s = f.truncate(s)
305         var quoted string
306         if f.sharp && strconv.CanBackquote(s) {
307             quoted = "`" + s + "`"
308         } else {
309             if f.plus {
310                 quoted = strconv.QuoteToASCII(s)
311             } else {
312                 quoted = strconv.Quote(s)
313             }
314         }
315         f.padString(quoted)
316     }
317
318     // fmt_qc formats the integer as a single-quoted, escaped Go
319     // If the character is not valid Unicode, it will print '\uf
320     func (f *fmt) fmt_qc(c int64) {
321         var quoted []byte
322         if f.plus {
323             quoted = strconv.AppendQuoteRuneToASCII(f.in
324         } else {
325             quoted = strconv.AppendQuoteRune(f.intbuf[0:
326         }
327         f.pad(quoted)
328     }
329
330     // floating-point
331
332     func doPrec(f *fmt, def int) int {
333         if f.precPresent {
334             return f.prec
335         }
336         return def
337     }
338
339     // formatFloat formats a float64; it is an efficient equival
340     func (f *fmt) formatFloat(v float64, verb byte, prec, n int)

```

```

341         // We leave one byte at the beginning of f.intbuf fo
342         // and make it a space, which we might be able to us
343         f.intbuf[0] = ' '
344         slice := strconv.AppendFloat(f.intbuf[0:1], v, verb,
345         // Add a plus sign or space to the floating-point st
346         // The formatted number starts at slice[1].
347         switch slice[1] {
348         case '-', '+':
349             // We're set; drop the leading space.
350             slice = slice[1:]
351         default:
352             // There's no sign, but we might need one.
353             if f.plus {
354                 slice[0] = '+'
355             } else if f.space {
356                 // space is already there
357             } else {
358                 slice = slice[1:]
359             }
360         }
361         f.pad(slice)
362     }
363
364     // fmt_e64 formats a float64 in the form -1.23e+12.
365     func (f *fmt) fmt_e64(v float64) { f.formatFloat(v, 'e', doP
366
367     // fmt_E64 formats a float64 in the form -1.23E+12.
368     func (f *fmt) fmt_E64(v float64) { f.formatFloat(v, 'E', doP
369
370     // fmt_f64 formats a float64 in the form -1.23.
371     func (f *fmt) fmt_f64(v float64) { f.formatFloat(v, 'f', doP
372
373     // fmt_g64 formats a float64 in the 'f' or 'e' form accordin
374     func (f *fmt) fmt_g64(v float64) { f.formatFloat(v, 'g', doP
375
376     // fmt_g64 formats a float64 in the 'f' or 'E' form accordin
377     func (f *fmt) fmt_G64(v float64) { f.formatFloat(v, 'G', doP
378
379     // fmt_fb64 formats a float64 in the form -123p3 (exponent i
380     func (f *fmt) fmt_fb64(v float64) { f.formatFloat(v, 'b', 0,
381
382     // float32
383     // cannot defer to float64 versions
384     // because it will get rounding wrong in corner cases.
385
386     // fmt_e32 formats a float32 in the form -1.23e+12.
387     func (f *fmt) fmt_e32(v float32) { f.formatFloat(float64(v),
388
389     // fmt_E32 formats a float32 in the form -1.23E+12.
390     func (f *fmt) fmt_E32(v float32) { f.formatFloat(float64(v),

```

```

391
392 // fmt_f32 formats a float32 in the form -1.23.
393 func (f *fmt) fmt_f32(v float32) { f.formatFloat(float64(v),
394
395 // fmt_g32 formats a float32 in the 'f' or 'e' form accordin
396 func (f *fmt) fmt_g32(v float32) { f.formatFloat(float64(v),
397
398 // fmt_G32 formats a float32 in the 'f' or 'E' form accordin
399 func (f *fmt) fmt_G32(v float32) { f.formatFloat(float64(v),
400
401 // fmt_fb32 formats a float32 in the form -123p3 (exponent i
402 func (f *fmt) fmt_fb32(v float32) { f.formatFloat(float64(v)
403
404 // fmt_c64 formats a complex64 according to the verb.
405 func (f *fmt) fmt_c64(v complex64, verb rune) {
406     f.buf.WriteByte('(')
407     r := real(v)
408     for i := 0; ; i++ {
409         switch verb {
410             case 'e':
411                 f.fmt_e32(r)
412             case 'E':
413                 f.fmt_E32(r)
414             case 'f':
415                 f.fmt_f32(r)
416             case 'g':
417                 f.fmt_g32(r)
418             case 'G':
419                 f.fmt_G32(r)
420         }
421         if i != 0 {
422             break
423         }
424         f.plus = true
425         r = imag(v)
426     }
427     f.buf.Write(irparenBytes)
428 }
429
430 // fmt_c128 formats a complex128 according to the verb.
431 func (f *fmt) fmt_c128(v complex128, verb rune) {
432     f.buf.WriteByte('(')
433     r := real(v)
434     for i := 0; ; i++ {
435         switch verb {
436             case 'e':
437                 f.fmt_e64(r)
438             case 'E':
439                 f.fmt_E64(r)

```

```
440         case 'f':
441             f.fmt_f64(r)
442         case 'g':
443             f.fmt_g64(r)
444         case 'G':
445             f.fmt_G64(r)
446         }
447         if i != 0 {
448             break
449         }
450         f.plus = true
451         r = imag(v)
452     }
453     f.buf.Write(irparenBytes)
454 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/fmt/print.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package fmt
6
7 import (
8     "errors"
9     "io"
10    "os"
11    "reflect"
12    "sync"
13    "unicode/utf8"
14 )
15
16 // Some constants in the form of bytes, to avoid string over
17 // Needlessly fastidious, I suppose.
18 var (
19     commaSpaceBytes = []byte(", ")
20     nilAngleBytes   = []byte("<nil>")
21     nilParenBytes   = []byte("(nil)")
22     nilBytes        = []byte("nil")
23     mapBytes        = []byte("map[")
24     missingBytes    = []byte("(MISSING)")
25     panicBytes      = []byte("(PANIC=")
26     extraBytes      = []byte("%!(EXTRA ")
27     irparenBytes    = []byte("i)")
28     bytesBytes      = []byte("[]byte{")
29     widthBytes      = []byte("%!(BADWIDTH)")
30     precBytes       = []byte("%!(BADPREC)")
31     noVerbBytes     = []byte("%!(NOVERB)")
32 )
33
34 // State represents the printer state passed to custom forma
35 // It provides access to the io.Writer interface plus inform
36 // the flags and options for the operand's format specifier.
37 type State interface {
38     // Write is the function to call to emit formatted o
39     Write(b []byte) (ret int, err error)
40     // Width returns the value of the width option and w
41     Width() (wid int, ok bool)
42     // Precision returns the value of the precision opti
43     Precision() (prec int, ok bool)
44 }
```

```

45         // Flag returns whether the flag c, a character, has
46         Flag(c int) bool
47     }
48
49     // Formatter is the interface implemented by values with a c
50     // The implementation of Format may call Sprintf or Fprintf(
51     // to generate its output.
52     type Formatter interface {
53         Format(f State, c rune)
54     }
55
56     // Stringer is implemented by any value that has a String me
57     // which defines the ``native'' format for that value.
58     // The String method is used to print values passed as an op
59     // to a %s or %v format or to an unformatted printer such as
60     type Stringer interface {
61         String() string
62     }
63
64     // GoStringer is implemented by any value that has a GoStrin
65     // which defines the Go syntax for that value.
66     // The GoString method is used to print values passed as an
67     // to a %#v format.
68     type GoStringer interface {
69         GoString() string
70     }
71
72     // Use simple []byte instead of bytes.Buffer to avoid large
73     type buffer []byte
74
75     func (b *buffer) Write(p []byte) (n int, err error) {
76         *b = append(*b, p...)
77         return len(p), nil
78     }
79
80     func (b *buffer) WriteString(s string) (n int, err error) {
81         *b = append(*b, s...)
82         return len(s), nil
83     }
84
85     func (b *buffer) WriteByte(c byte) error {
86         *b = append(*b, c)
87         return nil
88     }
89
90     func (bp *buffer) WriteRune(r rune) error {
91         if r < utf8.RuneSelf {
92             *bp = append(*bp, byte(r))
93             return nil
94         }

```

```

95
96     b := *bp
97     n := len(b)
98     for n+utf8.UTFMax > cap(b) {
99         b = append(b, 0)
100    }
101    w := utf8.EncodeRune(b[n:n+utf8.UTFMax], r)
102    *bp = b[:n+w]
103    return nil
104 }
105
106 type pp struct {
107     n          int
108     panicking  bool
109     erroring   bool // printing an error condition
110     buf        buffer
111     // field holds the current item, as an interface{}.
112     field      interface{}
113     // value holds the current item, as a reflect.Value,
114     // the zero Value if the item has not been reflected
115     value      reflect.Value
116     runeBuf    [utf8.UTFMax]byte
117     fmt        fmt
118 }
119
120 // A cache holds a set of reusable objects.
121 // The slice is a stack (LIFO).
122 // If more are needed, the cache creates them by calling new
123 type cache struct {
124     mu      sync.Mutex
125     saved   []interface{}
126     new     func() interface{}
127 }
128
129 func (c *cache) put(x interface{}) {
130     c.mu.Lock()
131     if len(c.saved) < cap(c.saved) {
132         c.saved = append(c.saved, x)
133     }
134     c.mu.Unlock()
135 }
136
137 func (c *cache) get() interface{} {
138     c.mu.Lock()
139     n := len(c.saved)
140     if n == 0 {
141         c.mu.Unlock()
142         return c.new()
143     }

```

```

144         x := c.saved[n-1]
145         c.saved = c.saved[0 : n-1]
146         c.mu.Unlock()
147         return x
148     }
149
150     func newCache(f func() interface{}) *cache {
151         return &cache{saved: make([]interface{}, 0, 100), ne
152     }
153
154     var ppFree = newCache(func() interface{} { return new(pp) })
155
156     // Allocate a new pp struct or grab a cached one.
157     func newPrinter() *pp {
158         p := ppFree.get().(*pp)
159         p.panicking = false
160         p.erroring = false
161         p.fmt.init(&p.buf)
162         return p
163     }
164
165     // Save used pp structs in ppFree; avoids an allocation per
166     func (p *pp) free() {
167         // Don't hold on to pp structs with large buffers.
168         if cap(p.buf) > 1024 {
169             return
170         }
171         p.buf = p.buf[:0]
172         p.field = nil
173         p.value = reflect.Value{}
174         ppFree.put(p)
175     }
176
177     func (p *pp) Width() (wid int, ok bool) { return p.fmt.wid,
178
179     func (p *pp) Precision() (prec int, ok bool) { return p.fmt.
180
181     func (p *pp) Flag(b int) bool {
182         switch b {
183             case '-':
184                 return p.fmt.minus
185             case '+':
186                 return p.fmt.plus
187             case '#':
188                 return p.fmt.sharp
189             case ' ':
190                 return p.fmt.space
191             case '0':
192                 return p.fmt.zero

```

```

193         }
194         return false
195     }
196
197     func (p *pp) add(c rune) {
198         p.buf.WriteRune(c)
199     }
200
201     // Implement Write so we can call Fprintf on a pp (through S
202     // recursive use in custom verbs.
203     func (p *pp) Write(b []byte) (ret int, err error) {
204         return p.buf.Write(b)
205     }
206
207     // These routines end in 'f' and take a format string.
208
209     // Fprintf formats according to a format specifier and write
210     // It returns the number of bytes written and any write error
211     func Fprintf(w io.Writer, format string, a ...interface{}) (
212         p := newPrinter()
213         p.doPrintf(format, a)
214         n64, err := w.Write(p.buf)
215         p.free()
216         return int(n64), err
217     }
218
219     // Printf formats according to a format specifier and writes
220     // It returns the number of bytes written and any write error
221     func Printf(format string, a ...interface{}) (n int, err error) {
222         return Fprintf(os.Stdout, format, a...)
223     }
224
225     // Sprintf formats according to a format specifier and return
226     func Sprintf(format string, a ...interface{}) string {
227         p := newPrinter()
228         p.doPrintf(format, a)
229         s := string(p.buf)
230         p.free()
231         return s
232     }
233
234     // Errorf formats according to a format specifier and return
235     // as a value that satisfies error.
236     func Errorf(format string, a ...interface{}) error {
237         return errors.New(Sprintf(format, a...))
238     }
239
240     // These routines do not take a format string
241
242     // Fprint formats using the default formats for its operands

```

```

243 // Spaces are added between operands when neither is a string
244 // It returns the number of bytes written and any write error
245 func Fprint(w io.Writer, a ...interface{}) (n int, err error) {
246     p := newPrinter()
247     p.doPrint(a, false, false)
248     n64, err := w.Write(p.buf)
249     p.free()
250     return int(n64), err
251 }
252
253 // Print formats using the default formats for its operands
254 // Spaces are added between operands when neither is a string
255 // It returns the number of bytes written and any write error
256 func Print(a ...interface{}) (n int, err error) {
257     return Fprint(os.Stdout, a...)
258 }
259
260 // Sprintf formats using the default formats for its operands
261 // Spaces are added between operands when neither is a string
262 func Sprintf(a ...interface{}) string {
263     p := newPrinter()
264     p.doPrint(a, false, false)
265     s := string(p.buf)
266     p.free()
267     return s
268 }
269
270 // These routines end in 'ln', do not take a format string,
271 // always add spaces between operands, and add a newline
272 // after the last operand.
273
274 // Fprintln formats using the default formats for its operands
275 // Spaces are always added between operands and a newline is
276 // It returns the number of bytes written and any write error
277 func Fprintln(w io.Writer, a ...interface{}) (n int, err error) {
278     p := newPrinter()
279     p.doPrint(a, true, true)
280     n64, err := w.Write(p.buf)
281     p.free()
282     return int(n64), err
283 }
284
285 // Println formats using the default formats for its operand
286 // Spaces are always added between operands and a newline is
287 // It returns the number of bytes written and any write error
288 func Println(a ...interface{}) (n int, err error) {
289     return Fprintln(os.Stdout, a...)
290 }
291

```

```

292 // Sprintln formats using the default formats for its operan
293 // Spaces are always added between operands and a newline is
294 func Sprintln(a ...interface{}) string {
295     p := newPrinter()
296     p.doPrint(a, true, true)
297     s := string(p.buf)
298     p.free()
299     return s
300 }
301
302 // Get the i'th arg of the struct value.
303 // If the arg itself is an interface, return a value for
304 // the thing inside the interface, not the interface itself.
305 func getField(v reflect.Value, i int) reflect.Value {
306     val := v.Field(i)
307     if val.Kind() == reflect.Interface && !val.IsNil() {
308         val = val.Elem()
309     }
310     return val
311 }
312
313 // Convert ASCII to integer. n is 0 (and got is false) if n
314 func parsenum(s string, start, end int) (num int, isnum bool)
315     if start >= end {
316         return 0, false, end
317     }
318     for newi = start; newi < end && '0' <= s[newi] && s[
319         num = num*10 + int(s[newi]-'0')
320         isnum = true
321     }
322     return
323 }
324
325 func (p *pp) unknownType(v interface{}) {
326     if v == nil {
327         p.buf.Write(nilAngleBytes)
328         return
329     }
330     p.buf.WriteByte('?')
331     p.buf.WriteString(reflect.TypeOf(v).String())
332     p.buf.WriteByte('?')
333 }
334
335 func (p *pp) badVerb(verb rune) {
336     p.erroring = true
337     p.add('%')
338     p.add('!')
339     p.add(verb)
340     p.add('(')

```

```

341     switch {
342     case p.field != nil:
343         p.buf.WriteString(reflect.TypeOf(p.field).St
344         p.add('=')
345         p.printField(p.field, 'v', false, false, 0)
346     case p.value.IsValid():
347         p.buf.WriteString(p.value.Type().String())
348         p.add('=')
349         p.printValue(p.value, 'v', false, false, 0)
350     default:
351         p.buf.Write(nilAngleBytes)
352     }
353     p.add(')')
354     p.erroring = false
355 }
356
357 func (p *pp) fmtBool(v bool, verb rune) {
358     switch verb {
359     case 't', 'v':
360         p.fmt.fmt_boolean(v)
361     default:
362         p.badVerb(verb)
363     }
364 }
365
366 // fmtC formats a rune for the 'c' format.
367 func (p *pp) fmtC(c int64) {
368     r := rune(c) // Check for overflow.
369     if int64(r) != c {
370         r = utf8.RuneError
371     }
372     w := utf8.EncodeRune(p.runeBuf[0:utf8.UTFMax], r)
373     p.fmt.pad(p.runeBuf[0:w])
374 }
375
376 func (p *pp) fmtInt64(v int64, verb rune) {
377     switch verb {
378     case 'b':
379         p.fmt.integer(v, 2, signed, ldigits)
380     case 'c':
381         p.fmtC(v)
382     case 'd', 'v':
383         p.fmt.integer(v, 10, signed, ldigits)
384     case 'o':
385         p.fmt.integer(v, 8, signed, ldigits)
386     case 'q':
387         if 0 <= v && v <= utf8.MaxRune {
388             p.fmt.fmt_qc(v)
389         } else {
390             p.badVerb(verb)

```

```

391     }
392     case 'x':
393         p.fmt.integer(v, 16, signed, ldigits)
394     case 'U':
395         p.fmtUnicode(v)
396     case 'X':
397         p.fmt.integer(v, 16, signed, udigits)
398     default:
399         p.badVerb(verb)
400     }
401 }
402
403 // fmt0x64 formats a uint64 in hexadecimal and prefixes it w
404 // not, as requested, by temporarily setting the sharp flag.
405 func (p *pp) fmt0x64(v uint64, leading0x bool) {
406     sharp := p.fmt.sharp
407     p.fmt.sharp = leading0x
408     p.fmt.integer(int64(v), 16, unsigned, ldigits)
409     p.fmt.sharp = sharp
410 }
411
412 // fmtUnicode formats a uint64 in U+1234 form by
413 // temporarily turning on the unicode flag and tweaking the
414 func (p *pp) fmtUnicode(v int64) {
415     precPresent := p.fmt.precPresent
416     sharp := p.fmt.sharp
417     p.fmt.sharp = false
418     prec := p.fmt.prec
419     if !precPresent {
420         // If prec is already set, leave it alone; o
421         p.fmt.prec = 4
422         p.fmt.precPresent = true
423     }
424     p.fmt.unicode = true // turn on U+
425     p.fmt.uniQuote = sharp
426     p.fmt.integer(int64(v), 16, unsigned, udigits)
427     p.fmt.unicode = false
428     p.fmt.uniQuote = false
429     p.fmt.prec = prec
430     p.fmt.precPresent = precPresent
431     p.fmt.sharp = sharp
432 }
433
434 func (p *pp) fmtUint64(v uint64, verb rune, goSyntax bool) {
435     switch verb {
436     case 'b':
437         p.fmt.integer(int64(v), 2, unsigned, ldigits)
438     case 'c':
439         p.fmtC(int64(v))

```

```

440     case 'd':
441         p.fmt.integer(int64(v), 10, unsigned, ldigit
442     case 'v':
443         if goSyntax {
444             p.fmt0x64(v, true)
445         } else {
446             p.fmt.integer(int64(v), 10, unsigned
447         }
448     case 'o':
449         p.fmt.integer(int64(v), 8, unsigned, ldigits
450     case 'q':
451         if 0 <= v && v <= utf8.MaxRune {
452             p.fmt.fmt_qc(int64(v))
453         } else {
454             p.badVerb(verb)
455         }
456     case 'x':
457         p.fmt.integer(int64(v), 16, unsigned, ldigit
458     case 'X':
459         p.fmt.integer(int64(v), 16, unsigned, udigit
460     case 'U':
461         p.fmtUnicode(int64(v))
462     default:
463         p.badVerb(verb)
464     }
465 }
466
467 func (p *pp) fmtFloat32(v float32, verb rune) {
468     switch verb {
469     case 'b':
470         p.fmt.fmt_fb32(v)
471     case 'e':
472         p.fmt.fmt_e32(v)
473     case 'E':
474         p.fmt.fmt_E32(v)
475     case 'f':
476         p.fmt.fmt_f32(v)
477     case 'g', 'v':
478         p.fmt.fmt_g32(v)
479     case 'G':
480         p.fmt.fmt_G32(v)
481     default:
482         p.badVerb(verb)
483     }
484 }
485
486 func (p *pp) fmtFloat64(v float64, verb rune) {
487     switch verb {
488     case 'b':

```

```

489         p.fmt.fmt_fb64(v)
490     case 'e':
491         p.fmt.fmt_e64(v)
492     case 'E':
493         p.fmt.fmt_E64(v)
494     case 'f':
495         p.fmt.fmt_f64(v)
496     case 'g', 'v':
497         p.fmt.fmt_g64(v)
498     case 'G':
499         p.fmt.fmt_G64(v)
500     default:
501         p.badVerb(verb)
502     }
503 }
504
505 func (p *pp) fmtComplex64(v complex64, verb rune) {
506     switch verb {
507     case 'e', 'E', 'f', 'F', 'g', 'G':
508         p.fmt.fmt_c64(v, verb)
509     case 'v':
510         p.fmt.fmt_c64(v, 'g')
511     default:
512         p.badVerb(verb)
513     }
514 }
515
516 func (p *pp) fmtComplex128(v complex128, verb rune) {
517     switch verb {
518     case 'e', 'E', 'f', 'F', 'g', 'G':
519         p.fmt.fmt_c128(v, verb)
520     case 'v':
521         p.fmt.fmt_c128(v, 'g')
522     default:
523         p.badVerb(verb)
524     }
525 }
526
527 func (p *pp) fmtString(v string, verb rune, goSyntax bool) {
528     switch verb {
529     case 'v':
530         if goSyntax {
531             p.fmt.fmt_q(v)
532         } else {
533             p.fmt.fmt_s(v)
534         }
535     case 's':
536         p.fmt.fmt_s(v)
537     case 'x':
538         p.fmt.fmt_sx(v, ldigits)

```

```

539         case 'X':
540             p.fmt.fmt_sx(v, udigits)
541         case 'q':
542             p.fmt.fmt_q(v)
543         default:
544             p.badVerb(verb)
545     }
546 }
547
548 func (p *pp) fmtBytes(v []byte, verb rune, goSyntax bool, de
549     if verb == 'v' || verb == 'd' {
550         if goSyntax {
551             p.buf.Write(bytesBytes)
552         } else {
553             p.buf.WriteByte('[')
554         }
555         for i, c := range v {
556             if i > 0 {
557                 if goSyntax {
558                     p.buf.Write(commaSpa
559                 } else {
560                     p.buf.WriteByte(' ')
561                 }
562             }
563             p.printField(c, 'v', p.fmt.plus, goS
564         }
565         if goSyntax {
566             p.buf.WriteByte('}')
567         } else {
568             p.buf.WriteByte(']')
569         }
570         return
571     }
572     s := string(v)
573     switch verb {
574     case 's':
575         p.fmt.fmt_s(s)
576     case 'x':
577         p.fmt.fmt_sx(s, ldigits)
578     case 'X':
579         p.fmt.fmt_sx(s, udigits)
580     case 'q':
581         p.fmt.fmt_q(s)
582     default:
583         p.badVerb(verb)
584     }
585 }
586
587 func (p *pp) fmtPointer(value reflect.Value, verb rune, goSy

```

```

588     switch verb {
589     case 'p', 'v', 'b', 'd', 'o', 'x', 'X':
590         // ok
591     default:
592         p.badVerb(verb)
593         return
594     }
595
596     var u uintptr
597     switch value.Kind() {
598     case reflect.Chan, reflect.Func, reflect.Map, reflect
599         u = value.Pointer()
600     default:
601         p.badVerb(verb)
602         return
603     }
604
605     if goSyntax {
606         p.add('(')
607         p.buf.WriteString(value.Type().String())
608         p.add(')')
609         p.add('(')
610         if u == 0 {
611             p.buf.Write(nilBytes)
612         } else {
613             p.fmt0x64(uint64(u), true)
614         }
615         p.add(')')
616     } else if verb == 'v' && u == 0 {
617         p.buf.Write(nilAngleBytes)
618     } else {
619         p.fmt0x64(uint64(u), !p.fmt.sharp)
620     }
621 }
622
623 var (
624     intBits      = reflect.TypeOf(0).Bits()
625     floatBits    = reflect.TypeOf(0.0).Bits()
626     complexBits = reflect.TypeOf(1i).Bits()
627     uintptrBits = reflect.TypeOf(uintptr(0)).Bits()
628 )
629
630 func (p *pp) catchPanic(field interface{}, verb rune) {
631     if err := recover(); err != nil {
632         // If it's a nil pointer, just say "<nil>".
633         // Stringer that fails to guard against nil
634         // value receiver, and in either case, "<nil
635         if v := reflect.ValueOf(field); v.Kind() ==
636             p.buf.Write(nilAngleBytes)

```

```

637         return
638     }
639     // Otherwise print a concise panic message.
640     // value will print itself nicely.
641     if p.panicking {
642         // Nested panics; the recursion in p
643         panic(err)
644     }
645     p.buf.WriteByte('%')
646     p.add(verb)
647     p.buf.Write(panicBytes)
648     p.panicking = true
649     p.printField(err, 'v', false, false, 0)
650     p.panicking = false
651     p.buf.WriteByte(')')
652 }
653 }
654
655 func (p *pp) handleMethods(verb rune, plus, goSyntax bool, d
656     if p.erroring {
657         return
658     }
659     // Is it a Formatter?
660     if formatter, ok := p.field.(Formatter); ok {
661         handled = true
662         wasString = false
663         defer p.catchPanic(p.field, verb)
664         formatter.Format(p, verb)
665         return
666     }
667     // Must not touch flags before Formatter looks at th
668     if plus {
669         p.fmt.plus = false
670     }
671
672     // If we're doing Go syntax and the field knows how
673     if goSyntax {
674         p.fmt.sharp = false
675         if stringer, ok := p.field.(GoStringer); ok
676             wasString = false
677             handled = true
678             defer p.catchPanic(p.field, verb)
679             // Print the result of GoString unad
680             p.fmtString(stringer.GoString(), 's'
681             return
682     }
683 } else {
684     // If a string is acceptable according to th
685     // the value satisfies one of the string-val
686     // Println etc. set verb to %v, which is "st

```

```

687         switch verb {
688         case 'v', 's', 'x', 'X', 'q':
689             // Is it an error or Stringer?
690             // The duplication in the bodies is
691             // setting wasString and handled, an
692             // must happen before calling the me
693             switch v := p.field.(type) {
694             case error:
695                 wasString = false
696                 handled = true
697                 defer p.catchPanic(p.field,
698                 p.printField(v.Error()), verb
699                 return
700
701             case Stringer:
702                 wasString = false
703                 handled = true
704                 defer p.catchPanic(p.field,
705                 p.printField(v.String()), ver
706                 return
707             }
708         }
709     }
710     handled = false
711     return
712 }
713
714 func (p *pp) printField(field interface{}, verb rune, plus,
715 if field == nil {
716     if verb == 'T' || verb == 'v' {
717         p.buf.Write(nilAngleBytes)
718     } else {
719         p.badVerb(verb)
720     }
721     return false
722 }
723
724 p.field = field
725 p.value = reflect.Value{}
726 // Special processing considerations.
727 // %T (the value's type) and %p (its address) are sp
728 switch verb {
729 case 'T':
730     p.printField(reflect.TypeOf(field).String(),
731     return false
732 case 'p':
733     p.fmtPointer(reflect.ValueOf(field), verb, g
734     return false
735 }

```

```

736
737     if wasString, handled := p.handleMethods(verb, plus,
738         return wasString
739     }
740
741     // Some types can be done without reflection.
742     switch f := field.(type) {
743     case bool:
744         p.fmtBool(f, verb)
745     case float32:
746         p.fmtFloat32(f, verb)
747     case float64:
748         p.fmtFloat64(f, verb)
749     case complex64:
750         p.fmtComplex64(complex64(f), verb)
751     case complex128:
752         p.fmtComplex128(f, verb)
753     case int:
754         p.fmtInt64(int64(f), verb)
755     case int8:
756         p.fmtInt64(int64(f), verb)
757     case int16:
758         p.fmtInt64(int64(f), verb)
759     case int32:
760         p.fmtInt64(int64(f), verb)
761     case int64:
762         p.fmtInt64(f, verb)
763     case uint:
764         p.fmtUint64(uint64(f), verb, goSyntax)
765     case uint8:
766         p.fmtUint64(uint64(f), verb, goSyntax)
767     case uint16:
768         p.fmtUint64(uint64(f), verb, goSyntax)
769     case uint32:
770         p.fmtUint64(uint64(f), verb, goSyntax)
771     case uint64:
772         p.fmtUint64(f, verb, goSyntax)
773     case uintptr:
774         p.fmtUint64(uint64(f), verb, goSyntax)
775     case string:
776         p.fmtString(f, verb, goSyntax)
777         wasString = verb == 's' || verb == 'v'
778     case []byte:
779         p.fmtBytes(f, verb, goSyntax, depth)
780         wasString = verb == 's'
781     default:
782         // Need to use reflection
783         return p.printReflectValue(reflect.ValueOf(f)
784     }

```

```

785         p.field = nil
786         return
787     }
788
789     // printValue is like printField but starts with a reflect v
790     func (p *pp) printValue(value reflect.Value, verb rune, plus
791         if !value.IsValid() {
792             if verb == 'T' || verb == 'v' {
793                 p.buf.Write(nilAngleBytes)
794             } else {
795                 p.badVerb(verb)
796             }
797             return false
798         }
799
800         // Special processing considerations.
801         // %T (the value's type) and %p (its address) are sp
802         switch verb {
803         case 'T':
804             p.printField(value.Type().String(), 's', fal
805             return false
806         case 'p':
807             p.fmtPointer(value, verb, goSyntax)
808             return false
809         }
810
811         // Handle values with special methods.
812         // Call always, even when field == nil, because hand
813         p.field = nil // Make sure it's cleared, for safety.
814         if value.CanInterface() {
815             p.field = value.Interface()
816         }
817         if wasString, handled := p.handleMethods(verb, plus,
818             return wasString
819         }
820
821         return p.printReflectValue(value, verb, plus, goSynt
822     }
823
824     // printReflectValue is the fallback for both printField and
825     // It uses reflect to print the value.
826     func (p *pp) printReflectValue(value reflect.Value, verb run
827         oldValue := p.value
828         p.value = value
829     BigSwitch:
830         switch f := value; f.Kind() {
831         case reflect.Bool:
832             p.fmtBool(f.Bool(), verb)
833         case reflect.Int, reflect.Int8, reflect.Int16, refle
834             p.fmtInt64(f.Int(), verb)

```

```

835 case reflect.Uint, reflect.Uint8, reflect.Uint16, re
836     p.fmtUint64(uint64(f.Uint()), verb, goSyntax
837 case reflect.Float32, reflect.Float64:
838     if f.Type().Size() == 4 {
839         p.fmtFloat32(float32(f.Float()), ver
840     } else {
841         p.fmtFloat64(float64(f.Float()), ver
842     }
843 case reflect.Complex64, reflect.Complex128:
844     if f.Type().Size() == 8 {
845         p.fmtComplex64(complex64(f.Complex())
846     } else {
847         p.fmtComplex128(complex128(f.Complex
848     }
849 case reflect.String:
850     p.fmtString(f.String(), verb, goSyntax)
851 case reflect.Map:
852     if goSyntax {
853         p.buf.WriteString(f.Type().String())
854         if f.IsNil() {
855             p.buf.WriteString("(nil)")
856             break
857         }
858         p.buf.WriteByte('{')
859     } else {
860         p.buf.Write(mapBytes)
861     }
862     keys := f.MapKeys()
863     for i, key := range keys {
864         if i > 0 {
865             if goSyntax {
866                 p.buf.Write(commaSpa
867             } else {
868                 p.buf.WriteByte(' ')
869             }
870         }
871         p.printValue(key, verb, plus, goSynt
872         p.buf.WriteByte(':')
873         p.printValue(f.MapIndex(key), verb,
874     }
875     if goSyntax {
876         p.buf.WriteByte('}')
877     } else {
878         p.buf.WriteByte(']')
879     }
880 case reflect.Struct:
881     if goSyntax {
882         p.buf.WriteString(value.Type().Strin
883     }

```

```

884         p.add('{')
885         v := f
886         t := v.Type()
887         for i := 0; i < v.NumField(); i++ {
888             if i > 0 {
889                 if goSyntax {
890                     p.buf.Write(commaSpa
891                 } else {
892                     p.buf.WriteByte(' ')
893                 }
894             }
895             if plus || goSyntax {
896                 if f := t.Field(i); f.Name != "" {
897                     p.buf.WriteString(f.Name)
898                     p.buf.WriteByte(':')
899                 }
900             }
901             p.printValue(getField(v, i), verb, p)
902         }
903         p.buf.WriteByte('}')
904     case reflect.Interface:
905         value := f.Elem()
906         if !value.IsValid() {
907             if goSyntax {
908                 p.buf.WriteString(f.Type().String())
909                 p.buf.Write(nilParenBytes)
910             } else {
911                 p.buf.Write(nilAngleBytes)
912             }
913         } else {
914             wasString = p.printValue(value, verb)
915         }
916     case reflect.Array, reflect.Slice:
917         // Byte slices are special.
918         if f.Type().Elem().Kind() == reflect.Uint8 {
919             // We know it's a slice of bytes, but
920             // []byte, or it would have been caught
921             // it directly in the (slightly) obvious
922             // that type, and we can't write an
923             // conversion because we don't have
924             // So we build a slice by hand. This
925             // if reflection could help a little
926             bytes := make([]byte, f.Len())
927             for i := range bytes {
928                 bytes[i] = byte(f.Index(i).Uint8)
929             }
930             p.fmtBytes(bytes, verb, goSyntax, de
931             wasString = verb == 's'
932             break

```

```

933     }
934     if goSyntax {
935         p.buf.WriteString(value.Type().Strin
936             if f.Kind() == reflect.Slice && f.Is
937                 p.buf.WriteString("(nil)")
938                 break
939             }
940         p.buf.WriteByte('{')
941     } else {
942         p.buf.WriteByte '[')
943     }
944     for i := 0; i < f.Len(); i++ {
945         if i > 0 {
946             if goSyntax {
947                 p.buf.Write(commaSpa
948             } else {
949                 p.buf.WriteByte(' ')
950             }
951         }
952         p.printValue(f.Index(i), verb, plus,
953     }
954     if goSyntax {
955         p.buf.WriteByte('}')
956     } else {
957         p.buf.WriteByte(']')
958     }
959 case reflect.Ptr:
960     v := f.Pointer()
961     // pointer to array or slice or struct? ok
962     // but not embedded (avoid loops)
963     if v != 0 && depth == 0 {
964         switch a := f.Elem(); a.Kind() {
965             case reflect.Array, reflect.Slice:
966                 p.buf.WriteByte('&')
967                 p.printValue(a, verb, plus,
968                 break BigSwitch
969             case reflect.Struct:
970                 p.buf.WriteByte('&')
971                 p.printValue(a, verb, plus,
972                 break BigSwitch
973         }
974     }
975     fallthrough
976 case reflect.Chan, reflect.Func, reflect.UnsafePoint
977     p.fmtPointer(value, verb, goSyntax)
978 default:
979     p.unknownType(f)
980 }
981 p.value = oldValue
982 return wasString

```

```

983 }
984
985 // intFromArg gets the fieldnumth element of a. On return, i
986 func intFromArg(a []interface{}, end, i, fieldnum int) (num
987     newi, newfieldnum = end, fieldnum
988     if i < end && fieldnum < len(a) {
989         num, isInt = a[fieldnum].(int)
990         newi, newfieldnum = i+1, fieldnum+1
991     }
992     return
993 }
994
995 func (p *pp) doPrintf(format string, a []interface{}) {
996     end := len(format)
997     fieldnum := 0 // we process one field per non-trivia
998     for i := 0; i < end; {
999         lasti := i
1000         for i < end && format[i] != '%' {
1001             i++
1002         }
1003         if i > lasti {
1004             p.buf.WriteString(format[lasti:i])
1005         }
1006         if i >= end {
1007             // done processing format string
1008             break
1009         }
1010
1011         // Process one verb
1012         i++
1013         // flags and widths
1014         p.fmt.clearflags()
1015         F:
1016         for ; i < end; i++ {
1017             switch format[i] {
1018                 case '#':
1019                     p.fmt.sharp = true
1020                 case '0':
1021                     p.fmt.zero = true
1022                 case '+':
1023                     p.fmt.plus = true
1024                 case '-':
1025                     p.fmt.minus = true
1026                 case ' ':
1027                     p.fmt.space = true
1028                 default:
1029                     break F
1030             }
1031         }

```

```

1032 // do we have width?
1033 if i < end && format[i] == '*' {
1034     p.fmt.wid, p.fmt.widPresent, i, fiel
1035     if !p.fmt.widPresent {
1036         p.buf.Write(widthBytes)
1037     }
1038 } else {
1039     p.fmt.wid, p.fmt.widPresent, i = par
1040 }
1041 // do we have precision?
1042 if i < end && format[i] == '.' {
1043     if format[i+1] == '*' {
1044         p.fmt.prec, p.fmt.precPresent
1045         if !p.fmt.precPresent {
1046             p.buf.Write(precByte
1047         }
1048     } else {
1049         p.fmt.prec, p.fmt.precPresent
1050         if !p.fmt.precPresent {
1051             p.fmt.prec = 0
1052             p.fmt.precPresent =
1053         }
1054     }
1055 }
1056 if i >= end {
1057     p.buf.Write(noVerbBytes)
1058     continue
1059 }
1060 c, w := utf8.DecodeRuneInString(format[i:])
1061 i += w
1062 // percent is special - absorbs no operand
1063 if c == '%' {
1064     p.buf.WriteByte('%') // We ignore wi
1065     continue
1066 }
1067 if fieldnum >= len(a) { // out of operands
1068     p.buf.WriteByte('%')
1069     p.add(c)
1070     p.buf.Write(missingBytes)
1071     continue
1072 }
1073 field := a[fieldnum]
1074 fieldnum++
1075
1076 goSyntax := c == 'v' && p.fmt.sharp
1077 plus := c == 'v' && p.fmt.plus
1078 p.printField(field, c, plus, goSyntax, 0)
1079 }
1080

```

```

1081         if fieldnum < len(a) {
1082             p.buf.Write(extraBytes)
1083             for ; fieldnum < len(a); fieldnum++ {
1084                 field := a[fieldnum]
1085                 if field != nil {
1086                     p.buf.WriteString(reflect.TypeName(field).String())
1087                     p.buf.WriteByte('=')
1088                 }
1089                 p.printField(field, 'v', false, false)
1090                 if fieldnum+1 < len(a) {
1091                     p.buf.Write(commaSpaceBytes)
1092                 }
1093             }
1094             p.buf.WriteByte(',')
1095         }
1096     }
1097
1098 func (p *pp) doPrint(a []interface{}, addspace, addnewline bool,
1099     prevString := false) {
1100     for fieldnum := 0; fieldnum < len(a); fieldnum++ {
1101         p.fmt.clearflags()
1102         // always add spaces if we're doing println
1103         field := a[fieldnum]
1104         if fieldnum > 0 {
1105             isString := field != nil && reflect.TypeOf(field).String() != ""
1106             if addspace || !isString && !prevString {
1107                 p.buf.WriteByte(' ')
1108             }
1109         }
1110         prevString = p.printField(field, 'v', false, false)
1111     }
1112     if addnewline {
1113         p.buf.WriteByte('\n')
1114     }
1115 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/fmt/scan.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package fmt
6
7 import (
8     "errors"
9     "io"
10    "math"
11    "os"
12    "reflect"
13    "strconv"
14    "unicode/utf8"
15 )
16
17 // runeUnreader is the interface to something that can unrea
18 // If the object provided to Scan does not satisfy this inte
19 // a local buffer will be used to back up the input, but its
20 // will be lost when Scan returns.
21 type runeUnreader interface {
22     UnreadRune() error
23 }
24
25 // ScanState represents the scanner state passed to custom s
26 // Scanners may do rune-at-a-time scanning or ask the ScanSt
27 // to discover the next space-delimited token.
28 type ScanState interface {
29     // ReadRune reads the next rune (Unicode code point)
30     // If invoked during Scanln, Fscanln, or Sscanln, Re
31     // return EOF after returning the first '\n' or when
32     // the specified width.
33     ReadRune() (r rune, size int, err error)
34     // UnreadRune causes the next call to ReadRune to re
35     UnreadRune() error
36     // SkipSpace skips space in the input. Newlines are
37     // unless the scan operation is Scanln, Fscanln or S
38     // a newline is treated as EOF.
39     SkipSpace()
40     // Token skips space in the input if skipSpace is tr
41     // run of Unicode code points c satisfying f(c). If
42     // !unicode.IsSpace(c) is used; that is, the token w
43     // characters. Newlines are treated as space unless
44     // is Scanln, Fscanln or Sscanln, in which case a ne
```

```

45         // EOF. The returned slice points to shared data th
46         // by the next call to Token, a call to a Scan funct
47         // as input, or when the calling Scan method returns
48         Token(skipSpace bool, f func(rune) bool) (token []by
49         // Width returns the value of the width option and w
50         // The unit is Unicode code points.
51         Width() (wid int, ok bool)
52         // Because ReadRune is implemented by the interface,
53         // called by the scanning routines and a valid imple
54         // ScanState may choose always to return an error fr
55         Read(buf []byte) (n int, err error)
56     }
57
58     // Scanner is implemented by any value that has a Scan metho
59     // the input for the representation of a value and stores th
60     // receiver, which must be a pointer to be useful. The Scan
61     // for any argument to Scan, Scanf, or Scanln that implement
62     type Scanner interface {
63         Scan(state ScanState, verb rune) error
64     }
65
66     // Scan scans text read from standard input, storing success
67     // space-separated values into successive arguments. Newlin
68     // as space. It returns the number of items successfully sc
69     // If that is less than the number of arguments, err will re
70     func Scan(a ...interface{}) (n int, err error) {
71         return Fscan(os.Stdin, a...)
72     }
73
74     // Scanln is similar to Scan, but stops scanning at a newlin
75     // after the final item there must be a newline or EOF.
76     func Scanln(a ...interface{}) (n int, err error) {
77         return Fscanln(os.Stdin, a...)
78     }
79
80     // Scanf scans text read from standard input, storing succes
81     // space-separated values into successive arguments as deter
82     // the format. It returns the number of items successfully
83     func Scanf(format string, a ...interface{}) (n int, err erro
84         return Fscanf(os.Stdin, format, a...)
85     }
86
87     type stringReader string
88
89     func (r *stringReader) Read(b []byte) (n int, err error) {
90         n = copy(b, *r)
91         *r = (*r)[n:]
92         if n == 0 {
93             err = io.EOF
94         }

```

```

95         return
96     }
97
98     // Sscan scans the argument string, storing successive space
99     // values into successive arguments. Newlines count as spac
100    // returns the number of items successfully scanned. If tha
101    // than the number of arguments, err will report why.
102    func Sscan(str string, a ...interface{}) (n int, err error)
103        return Fscan((*stringReader)(&str), a...)
104    }
105
106    // Sscanln is similar to Sscan, but stops scanning at a newl
107    // after the final item there must be a newline or EOF.
108    func Sscanln(str string, a ...interface{}) (n int, err error)
109        return Fscanln((*stringReader)(&str), a...)
110    }
111
112    // Sscanf scans the argument string, storing successive spac
113    // values into successive arguments as determined by the for
114    // returns the number of items successfully parsed.
115    func Sscanf(str string, format string, a ...interface{}) (n
116        return Fscanf((*stringReader)(&str), format, a...)
117    }
118
119    // Fscan scans text read from r, storing successive space-se
120    // values into successive arguments. Newlines count as spac
121    // returns the number of items successfully scanned. If tha
122    // than the number of arguments, err will report why.
123    func Fscan(r io.Reader, a ...interface{}) (n int, err error)
124        s, old := newScanState(r, true, false)
125        n, err = s.doScan(a)
126        s.free(old)
127        return
128    }
129
130    // Fscanln is similar to Fscan, but stops scanning at a newl
131    // after the final item there must be a newline or EOF.
132    func Fscanln(r io.Reader, a ...interface{}) (n int, err erro
133        s, old := newScanState(r, false, true)
134        n, err = s.doScan(a)
135        s.free(old)
136        return
137    }
138
139    // Fscanf scans text read from r, storing successive space-s
140    // values into successive arguments as determined by the for
141    // returns the number of items successfully parsed.
142    func Fscanf(r io.Reader, format string, a ...interface{}) (n
143        s, old := newScanState(r, false, false)

```

```

144         n, err = s.doScanf(format, a)
145         s.free(old)
146         return
147     }
148
149     // scanError represents an error generated by the scanning s
150     // It's used as a unique signature to identify such errors w
151     type scanError struct {
152         err error
153     }
154
155     const eof = -1
156
157     // ss is the internal implementation of ScanState.
158     type ss struct {
159         rr      io.RuneReader // where to read input
160         buf      buffer      // token accumulator
161         peekRune rune      // one-rune lookahead
162         prevRune rune      // last rune returned by Read
163         count   int       // runes consumed so far.
164         atEOF   bool      // already read EOF
165         ssave
166     }
167
168     // ssave holds the parts of ss that need to be
169     // saved and restored on recursive scans.
170     type ssave struct {
171         validSave bool // is or was a part of an actual ss.
172         nlIsEnd   bool // whether newline terminates scan
173         nlIsSpace bool // whether newline counts as white s
174         fieldLimit int // max value of ss.count for this fi
175         limit     int // max value of ss.count.
176         maxWid   int // width of this field.
177     }
178
179     // The Read method is only in ScanState so that ScanState
180     // satisfies io.Reader. It will never be called when used as
181     // intended, so there is no need to make it actually work.
182     func (s *ss) Read(buf []byte) (n int, err error) {
183         return 0, errors.New("ScanState's Read should not be
184     }
185
186     func (s *ss) ReadRune() (r rune, size int, err error) {
187         if s.peekRune >= 0 {
188             s.count++
189             r = s.peekRune
190             size = utf8.RuneLen(r)
191             s.prevRune = r
192             s.peekRune = -1

```

```

193         return
194     }
195     if s.atEOF || s.n1IsEnd && s.prevRune == '\n' || s.c
196         err = io.EOF
197         return
198     }
199
200     r, size, err = s.rr.ReadRune()
201     if err == nil {
202         s.count++
203         s.prevRune = r
204     } else if err == io.EOF {
205         s.atEOF = true
206     }
207     return
208 }
209
210 func (s *ss) Width() (wid int, ok bool) {
211     if s.maxWid == hugeWid {
212         return 0, false
213     }
214     return s.maxWid, true
215 }
216
217 // The public method returns an error; this private one pani
218 // If getRune reaches EOF, the return value is EOF (-1).
219 func (s *ss) getRune() (r rune) {
220     r, _, err := s.ReadRune()
221     if err != nil {
222         if err == io.EOF {
223             return eof
224         }
225         s.error(err)
226     }
227     return
228 }
229
230 // mustReadRune turns io.EOF into a panic(io.ErrUnexpectedEO
231 // It is called in cases such as string scanning where an EO
232 // syntax error.
233 func (s *ss) mustReadRune() (r rune) {
234     r = s.getRune()
235     if r == eof {
236         s.error(io.ErrUnexpectedEOF)
237     }
238     return
239 }
240
241 func (s *ss) UnreadRune() error {
242     if u, ok := s.rr.(runeUnreader); ok {

```

```

243         u.UnreadRune()
244     } else {
245         s.peekRune = s.prevRune
246     }
247     s.prevRune = -1
248     s.count--
249     return nil
250 }
251
252 func (s *ss) error(err error) {
253     panic(scanError{err})
254 }
255
256 func (s *ss) errorString(err string) {
257     panic(scanError{errors.New(err)})
258 }
259
260 func (s *ss) Token(skipSpace bool, f func(rune) bool) (tok [
261     defer func() {
262         if e := recover(); e != nil {
263             if se, ok := e.(scanError); ok {
264                 err = se.err
265             } else {
266                 panic(e)
267             }
268         }
269     }()
270     if f == nil {
271         f = notSpace
272     }
273     s.buf = s.buf[:0]
274     tok = s.token(skipSpace, f)
275     return
276 }
277
278 // space is a copy of the unicode.White_Space ranges,
279 // to avoid depending on package unicode.
280 var space = [][]uint16{
281     {0x0009, 0x000d},
282     {0x0020, 0x0020},
283     {0x0085, 0x0085},
284     {0x00a0, 0x00a0},
285     {0x1680, 0x1680},
286     {0x180e, 0x180e},
287     {0x2000, 0x200a},
288     {0x2028, 0x2029},
289     {0x202f, 0x202f},
290     {0x205f, 0x205f},
291     {0x3000, 0x3000},

```

```

292 }
293
294 func isSpace(r rune) bool {
295     if r >= 1<<16 {
296         return false
297     }
298     rx := uint16(r)
299     for _, rng := range space {
300         if rx < rng[0] {
301             return false
302         }
303         if rx <= rng[1] {
304             return true
305         }
306     }
307     return false
308 }
309
310 // notSpace is the default scanning function used in Token.
311 func notSpace(r rune) bool {
312     return !isSpace(r)
313 }
314
315 // skipSpace provides Scan() methods the ability to skip spa
316 // in keeping with the current scanning mode set by format s
317 func (s *ss) SkipSpace() {
318     s.skipSpace(false)
319 }
320
321 // readRune is a structure to enable reading UTF-8 encoded c
322 // from an io.Reader. It is used if the Reader given to the
323 // not already implement io.RuneReader.
324 type readRune struct {
325     reader io.Reader
326     buf    [utf8.UTFMax]byte // used only inside ReadRu
327     pending int                // number of bytes in pend
328     pendBuf [utf8.UTFMax]byte // bytes left over
329 }
330
331 // readByte returns the next byte from the input, which may
332 // left over from a previous read if the UTF-8 was ill-forme
333 func (r *readRune) readByte() (b byte, err error) {
334     if r.pending > 0 {
335         b = r.pendBuf[0]
336         copy(r.pendBuf[0:], r.pendBuf[1:])
337         r.pending--
338         return
339     }
340     _, err = r.reader.Read(r.pendBuf[0:1])

```

```

341         return r.pendBuf[0], err
342     }
343
344     // unread saves the bytes for the next read.
345     func (r *readRune) unread(buf []byte) {
346         copy(r.pendBuf[r.pending:], buf)
347         r.pending += len(buf)
348     }
349
350     // ReadRune returns the next UTF-8 encoded code point from t
351     // io.Reader inside r.
352     func (r *readRune) ReadRune() (rr rune, size int, err error) {
353         r.buf[0], err = r.readByte()
354         if err != nil {
355             return 0, 0, err
356         }
357         if r.buf[0] < utf8.RuneSelf { // fast check for comm
358             rr = rune(r.buf[0])
359             return
360         }
361         var n int
362         for n = 1; !utf8.FullRune(r.buf[0:n]); n++ {
363             r.buf[n], err = r.readByte()
364             if err != nil {
365                 if err == io.EOF {
366                     err = nil
367                     break
368                 }
369                 return
370             }
371         }
372         rr, size = utf8.DecodeRune(r.buf[0:n])
373         if size < n { // an error
374             r.unread(r.buf[size:n])
375         }
376         return
377     }
378
379     var ssFree = newCache(func() interface{} { return new(ss) })
380
381     // Allocate a new ss struct or grab a cached one.
382     func newScanState(r io.Reader, nlIsSpace, nlIsEnd bool) (s *
383         // If the reader is a *ss, then we've got a recursiv
384         // call to Scan, so re-use the scan state.
385         s, ok := r.(*ss)
386         if ok {
387             old = s.ssave
388             s.limit = s.fieldLimit
389             s.nlIsEnd = nlIsEnd || s.nlIsEnd
390             s.nlIsSpace = nlIsSpace

```

```

391         return
392     }
393
394     s = ssFree.get().(*ss)
395     if rr, ok := r.(io.RuneReader); ok {
396         s.rr = rr
397     } else {
398         s.rr = &readRune{reader: r}
399     }
400     s.nlIsSpace = nlIsSpace
401     s.nlIsEnd = nlIsEnd
402     s.prevRune = -1
403     s.peekRune = -1
404     s.atEOF = false
405     s.limit = hugeWid
406     s.fieldLimit = hugeWid
407     s.maxWid = hugeWid
408     s.validSave = true
409     s.count = 0
410     return
411 }
412
413 // Save used ss structs in ssFree; avoid an allocation per i
414 func (s *ss) free(old ssave) {
415     // If it was used recursively, just restore the old
416     if old.validSave {
417         s.ssave = old
418         return
419     }
420     // Don't hold on to ss structs with large buffers.
421     if cap(s.buf) > 1024 {
422         return
423     }
424     s.buf = s.buf[:0]
425     s.rr = nil
426     ssFree.put(s)
427 }
428
429 // skipSpace skips spaces and maybe newlines.
430 func (s *ss) skipSpace(stopAtNewline bool) {
431     for {
432         r := s.getRune()
433         if r == eof {
434             return
435         }
436         if r == '\n' {
437             if stopAtNewline {
438                 break
439             }

```

```

440             if s.nlIsSpace {
441                 continue
442             }
443             s.errorString("unexpected newline")
444             return
445         }
446         if !isSpace(r) {
447             s.UnreadRune()
448             break
449         }
450     }
451 }
452
453 // token returns the next space-delimited string from the in
454 // skips white space. For Scanln, it stops at newlines. For
455 // newlines are treated as spaces.
456 func (s *ss) token(skipSpace bool, f func(rune) bool) []byte
457     if skipSpace {
458         s.skipSpace(false)
459     }
460     // read until white space or newline
461     for {
462         r := s.getRune()
463         if r == eof {
464             break
465         }
466         if !f(r) {
467             s.UnreadRune()
468             break
469         }
470         s.buf.WriteRune(r)
471     }
472     return s.buf
473 }
474
475 // typeError indicates that the type of the operand did not
476 func (s *ss) typeError(field interface{}, expected string) {
477     s.errorString("expected field of type pointer to " +
478 }
479
480 var complexError = errors.New("syntax error scanning complex")
481 var boolError = errors.New("syntax error scanning boolean")
482
483 func indexRune(s string, r rune) int {
484     for i, c := range s {
485         if c == r {
486             return i
487         }
488     }

```

```

489         return -1
490     }
491
492     // consume reads the next rune in the input and reports whet
493     // If accept is true, it puts the character into the input t
494     func (s *ss) consume(ok string, accept bool) bool {
495         r := s.getRune()
496         if r == eof {
497             return false
498         }
499         if indexRune(ok, r) >= 0 {
500             if accept {
501                 s.buf.WriteRune(r)
502             }
503             return true
504         }
505         if r != eof && accept {
506             s.UnreadRune()
507         }
508         return false
509     }
510
511     // peek reports whether the next character is in the ok stri
512     func (s *ss) peek(ok string) bool {
513         r := s.getRune()
514         if r != eof {
515             s.UnreadRune()
516         }
517         return indexRune(ok, r) >= 0
518     }
519
520     func (s *ss) notEOF() {
521         // Guarantee there is data to be read.
522         if r := s.getRune(); r == eof {
523             panic(io.EOF)
524         }
525         s.UnreadRune()
526     }
527
528     // accept checks the next rune in the input. If it's a byte
529     // buffer and returns true. Otherwise it return false.
530     func (s *ss) accept(ok string) bool {
531         return s.consume(ok, true)
532     }
533
534     // okVerb verifies that the verb is present in the list, set
535     func (s *ss) okVerb(verb rune, okVerbs, typ string) bool {
536         for _, v := range okVerbs {
537             if v == verb {
538                 return true

```

```

539         }
540     }
541     s.errorString("bad verb %" + string(verb) + " for ")
542     return false
543 }
544
545 // scanBool returns the value of the boolean represented by
546 func (s *ss) scanBool(verb rune) bool {
547     s.skipSpace(false)
548     s.notEOF()
549     if !s.okVerb(verb, "tv", "boolean") {
550         return false
551     }
552     // Syntax-checking a boolean is annoying. We're not
553     switch s.getRune() {
554     case '0':
555         return false
556     case '1':
557         return true
558     case 't', 'T':
559         if s.accept("rR") && (!s.accept("uU") || !s.
560             s.error(boolError)
561         }
562         return true
563     case 'f', 'F':
564         if s.accept("aA") && (!s.accept("lL") || !s.
565             s.error(boolError)
566         }
567         return false
568     }
569     return false
570 }
571
572 // Numerical elements
573 const (
574     binaryDigits      = "01"
575     octalDigits       = "01234567"
576     decimalDigits     = "0123456789"
577     hexadecimalDigits = "0123456789aAbBcCdDeEfF"
578     sign              = "+-"
579     period            = "."
580     exponent          = "eEp"
581 )
582
583 // getBase returns the numeric base represented by the verb
584 func (s *ss) getBase(verb rune) (base int, digits string) {
585     s.okVerb(verb, "bdoUxV", "integer") // sets s.err
586     base = 10
587     digits = decimalDigits

```

```

588         switch verb {
589             case 'b':
590                 base = 2
591                 digits = binaryDigits
592             case 'o':
593                 base = 8
594                 digits = octalDigits
595             case 'x', 'X', 'U':
596                 base = 16
597                 digits = hexadecimalDigits
598         }
599         return
600     }
601
602 // scanNumber returns the numerical string with specified di
603 func (s *ss) scanNumber(digits string, haveDigits bool) stri
604     if !haveDigits {
605         s.notEOF()
606         if !s.accept(digits) {
607             s.errorString("expected integer")
608         }
609     }
610     for s.accept(digits) {
611     }
612     return string(s.buf)
613 }
614
615 // scanRune returns the next rune value in the input.
616 func (s *ss) scanRune(bitSize int) int64 {
617     s.notEOF()
618     r := int64(s.getRune())
619     n := uint(bitSize)
620     x := (r << (64 - n)) >> (64 - n)
621     if x != r {
622         s.errorString("overflow on character value ")
623     }
624     return r
625 }
626
627 // scanBasePrefix reports whether the integer begins with a
628 // and returns the base, digit string, and whether a zero wa
629 // It is called only if the verb is %v.
630 func (s *ss) scanBasePrefix() (base int, digits string, foun
631     if !s.peek("0") {
632         return 10, decimalDigits, false
633     }
634     s.accept("0")
635     found = true // We've put a digit into the token buf
636     // Special cases for '0' && '0x'

```

```

637         base, digits = 8, octalDigits
638         if s.peek("xX") {
639             s.consume("xX", false)
640             base, digits = 16, hexadecimalDigits
641         }
642         return
643     }
644
645     // scanInt returns the value of the integer represented by t
646     // token, checking for overflow. Any error is stored in s.e
647     func (s *ss) scanInt(verb rune, bitSize int) int64 {
648         if verb == 'c' {
649             return s.scanRune(bitSize)
650         }
651         s.skipSpace(false)
652         s.notEOF()
653         base, digits := s.getBase(verb)
654         haveDigits := false
655         if verb == 'U' {
656             if !s.consume("U", false) || !s.consume("+",
657                 s.errorString("bad unicode format "))
658         }
659         } else {
660             s.accept(sign) // If there's a sign, it will
661             if verb == 'v' {
662                 base, digits, haveDigits = s.scanBas
663             }
664         }
665         tok := s.scanNumber(digits, haveDigits)
666         i, err := strconv.ParseInt(tok, base, 64)
667         if err != nil {
668             s.error(err)
669         }
670         n := uint(bitSize)
671         x := (i << (64 - n)) >> (64 - n)
672         if x != i {
673             s.errorString("integer overflow on token " +
674                 )
675         }
676         return i
677     }
678
679     // scanUint returns the value of the unsigned integer repres
680     // by the next token, checking for overflow. Any error is s
681     func (s *ss) scanUint(verb rune, bitSize int) uint64 {
682         if verb == 'c' {
683             return uint64(s.scanRune(bitSize))
684         }
685         s.skipSpace(false)
686         s.notEOF()
687         base, digits := s.getBase(verb)

```

```

687     haveDigits := false
688     if verb == 'U' {
689         if !s.consume("U", false) || !s.consume("+",
690             s.errorString("bad unicode format "))
691     }
692     } else if verb == 'v' {
693         base, digits, haveDigits = s.scanBasePrefix(
694     }
695     tok := s.scanNumber(digits, haveDigits)
696     i, err := strconv.ParseUint(tok, base, 64)
697     if err != nil {
698         s.error(err)
699     }
700     n := uint(bitSize)
701     x := (i << (64 - n)) >> (64 - n)
702     if x != i {
703         s.errorString("unsigned integer overflow on
704     }
705     return i
706 }
707
708 // floatToken returns the floating-point number starting her
709 // if the width is specified. It's not rigorous about syntax
710 // we have at least some digits, but Atof will do that.
711 func (s *ss) floatToken() string {
712     s.buf = s.buf[:0]
713     // NaN?
714     if s.accept("nN") && s.accept("aA") && s.accept("nN"
715         return string(s.buf)
716     }
717     // leading sign?
718     s.accept(sign)
719     // Inf?
720     if s.accept("iI") && s.accept("nN") && s.accept("fF"
721         return string(s.buf)
722     }
723     // digits?
724     for s.accept(decimalDigits) {
725     }
726     // decimal point?
727     if s.accept(period) {
728         // fraction?
729         for s.accept(decimalDigits) {
730         }
731     }
732     // exponent?
733     if s.accept(exponent) {
734         // leading sign?
735         s.accept(sign)

```



```

785             }
786             s.error(err)
787         }
788         return math.Ldexp(f, n)
789     }
790     f, err := strconv.ParseFloat(str, n)
791     if err != nil {
792         s.error(err)
793     }
794     return f
795 }
796
797 // convertComplex converts the next token to a complex128 va
798 // The atof argument is a type-specific reader for the under
799 // If we're reading complex64, atof will parse float32s and
800 // to float64's to avoid reproducing this code for each comp
801 func (s *ss) scanComplex(verb rune, n int) complex128 {
802     if !s.okVerb(verb, floatVerbs, "complex") {
803         return 0
804     }
805     s.skipSpace(false)
806     s.notEOF()
807     sreal, simag := s.complexTokens()
808     real := s.convertFloat(sreal, n/2)
809     imag := s.convertFloat(simag, n/2)
810     return complex(real, imag)
811 }
812
813 // convertString returns the string represented by the next
814 // The format of the input is determined by the verb.
815 func (s *ss) convertString(verb rune) (str string) {
816     if !s.okVerb(verb, "svqx", "string") {
817         return ""
818     }
819     s.skipSpace(false)
820     s.notEOF()
821     switch verb {
822     case 'q':
823         str = s.quotedString()
824     case 'x':
825         str = s.hexString()
826     default:
827         str = string(s.token(true, notSpace)) // %s
828     }
829     return
830 }
831
832 // quotedString returns the double- or back-quoted string re
833 func (s *ss) quotedString() string {
834     s.notEOF()

```

```

835     quote := s.getRune()
836     switch quote {
837     case '`':
838         // Back-quoted: Anything goes until EOF or b
839         for {
840             r := s.mustReadRune()
841             if r == quote {
842                 break
843             }
844             s.buf.WriteRune(r)
845         }
846         return string(s.buf)
847     case '"':
848         // Double-quoted: Include the quotes and let
849         s.buf.WriteRune(quote)
850         for {
851             r := s.mustReadRune()
852             s.buf.WriteRune(r)
853             if r == '\\\ ' {
854                 // In a legal backslash esca
855                 // immediately after the esc
856                 // Thus we only need to prot
857                 r := s.mustReadRune()
858                 s.buf.WriteRune(r)
859             } else if r == '"' {
860                 break
861             }
862         }
863         result, err := strconv.Unquote(string(s.buf))
864         if err != nil {
865             s.error(err)
866         }
867         return result
868     default:
869         s.errorString("expected quoted string")
870     }
871     return ""
872 }
873
874 // hexDigit returns the value of the hexadecimal digit
875 func (s *ss) hexDigit(d rune) int {
876     digit := int(d)
877     switch digit {
878     case '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
879         return digit - '0'
880     case 'a', 'b', 'c', 'd', 'e', 'f':
881         return 10 + digit - 'a'
882     case 'A', 'B', 'C', 'D', 'E', 'F':
883         return 10 + digit - 'A'

```

```

884     }
885     s.errorString("Scan: illegal hex digit")
886     return 0
887 }
888
889 // hexByte returns the next hex-encoded (two-character) byte
890 // There must be either two hexadecimal digits or a space ch
891 func (s *ss) hexByte() (b byte, ok bool) {
892     rune1 := s.getRune()
893     if rune1 == eof {
894         return
895     }
896     if isSpace(rune1) {
897         s.UnreadRune()
898         return
899     }
900     rune2 := s.mustReadRune()
901     return byte(s.hexDigit(rune1)<<4 | s.hexDigit(rune2))
902 }
903
904 // hexString returns the space-delimited hexpair-encoded str
905 func (s *ss) hexString() string {
906     s.notEOF()
907     for {
908         b, ok := s.hexByte()
909         if !ok {
910             break
911         }
912         s.buf.WriteByte(b)
913     }
914     if len(s.buf) == 0 {
915         s.errorString("Scan: no hex data for %x stri
916         return ""
917     }
918     return string(s.buf)
919 }
920
921 const floatVerbs = "beEfFgGv"
922
923 const hugeWid = 1 << 30
924
925 // scanOne scans a single value, deriving the scanner from t
926 func (s *ss) scanOne(verb rune, field interface{}) {
927     s.buf = s.buf[:0]
928     var err error
929     // If the parameter has its own Scan method, use tha
930     if v, ok := field.(Scanner); ok {
931         err = v.Scan(s, verb)
932         if err != nil {

```

```

933         if err == io.EOF {
934             err = io.ErrUnexpectedEOF
935         }
936         s.error(err)
937     }
938     return
939 }
940
941 switch v := field.(type) {
942 case *bool:
943     *v = s.scanBool(verb)
944 case *complex64:
945     *v = complex64(s.scanComplex(verb, 64))
946 case *complex128:
947     *v = s.scanComplex(verb, 128)
948 case *int:
949     *v = int(s.scanInt(verb, intBits))
950 case *int8:
951     *v = int8(s.scanInt(verb, 8))
952 case *int16:
953     *v = int16(s.scanInt(verb, 16))
954 case *int32:
955     *v = int32(s.scanInt(verb, 32))
956 case *int64:
957     *v = s.scanInt(verb, 64)
958 case *uint:
959     *v = uint(s.scanUint(verb, intBits))
960 case *uint8:
961     *v = uint8(s.scanUint(verb, 8))
962 case *uint16:
963     *v = uint16(s.scanUint(verb, 16))
964 case *uint32:
965     *v = uint32(s.scanUint(verb, 32))
966 case *uint64:
967     *v = s.scanUint(verb, 64)
968 case *uintptr:
969     *v = uintptr(s.scanUint(verb, uintptrBits))
970 // Floats are tricky because you want to scan in the
971 // scan in high precision and convert, in order to p
972 case *float32:
973     if s.okVerb(verb, floatVerbs, "float32") {
974         s.skipSpace(false)
975         s.notEOF()
976         *v = float32(s.convertFloat(s.floatT
977     }
978 case *float64:
979     if s.okVerb(verb, floatVerbs, "float64") {
980         s.skipSpace(false)
981         s.notEOF()
982         *v = s.convertFloat(s.floatToken(),

```

```

983     }
984     case *string:
985         *v = s.convertString(verb)
986     case *[]byte:
987         // We scan to string and convert so we get a
988         // If we scanned to bytes, the slice would p
989         *v = []byte(s.convertString(verb))
990     default:
991         val := reflect.ValueOf(v)
992         ptr := val
993         if ptr.Kind() != reflect.Ptr {
994             s.errorString("Scan: type not a poin
995             return
996         }
997         switch v := ptr.Elem(); v.Kind() {
998         case reflect.Bool:
999             v.SetBool(s.scanBool(verb))
1000        case reflect.Int, reflect.Int8, reflect.Int1
1001            v.SetInt(s.scanInt(verb, v.Type()).Bi
1002        case reflect.Uint, reflect.Uint8, reflect.Ui
1003            v.SetUint(s.scanUint(verb, v.Type()).
1004        case reflect.String:
1005            v.SetString(s.convertString(verb))
1006        case reflect.Slice:
1007            // For now, can only handle (renamed
1008            typ := v.Type()
1009            if typ.Elem().Kind() != reflect.Uint
1010                s.errorString("Scan: can't h
1011            }
1012            str := s.convertString(verb)
1013            v.Set(reflect.MakeSlice(typ, len(str)
1014            for i := 0; i < len(str); i++ {
1015                v.Index(i).SetUint(uint64(st
1016            }
1017        case reflect.Float32, reflect.Float64:
1018            s.skipSpace(false)
1019            s.notEOF()
1020            v.SetFloat(s.convertFloat(s.floatTok
1021        case reflect.Complex64, reflect.Complex128:
1022            v.SetComplex(s.scanComplex(verb, v.T
1023        default:
1024            s.errorString("Scan: can't handle ty
1025        }
1026    }
1027 }
1028
1029 // errorHandler turns local panics into error returns.
1030 func errorHandler(errp *error) {
1031     if e := recover(); e != nil {

```

```

1032         if se, ok := e.(scanError); ok { // catch lo
1033             *errp = se.err
1034         } else if eof, ok := e.(error); ok && eof ==
1035             *errp = eof
1036         } else {
1037             panic(e)
1038         }
1039     }
1040 }
1041
1042 // doScan does the real work for scanning without a format s
1043 func (s *ss) doScan(a []interface{}) (numProcessed int, err
1044     defer errorHandler(&err)
1045     for _, field := range a {
1046         s.scanOne('v', field)
1047         numProcessed++
1048     }
1049     // Check for newline if required.
1050     if !s.nlIsSpace {
1051         for {
1052             r := s.getRune()
1053             if r == '\n' || r == eof {
1054                 break
1055             }
1056             if !isSpace(r) {
1057                 s.errorString("Scan: expecte
1058                 break
1059             }
1060         }
1061     }
1062     return
1063 }
1064
1065 // advance determines whether the next characters in the inp
1066 // those of the format. It returns the number of bytes (sic
1067 // in the format. Newlines included, all runs of space chara
1068 // either input or format behave as a single space. This rou
1069 // handles the %% case. If the return value is zero, either
1070 // starts with a % (with no following %) or the input is emp
1071 // If it is negative, the input did not match the string.
1072 func (s *ss) advance(format string) (i int) {
1073     for i < len(format) {
1074         fmtc, w := utf8.DecodeRuneInString(format[i:
1075         if fmtc == '%' {
1076             // %% acts like a real percent
1077             nextc, _ := utf8.DecodeRuneInString(
1078             if nextc != '%' {
1079                 return
1080             }

```

```

1081         i += w // skip the first %
1082     }
1083     sawSpace := false
1084     for isSpace(fmtc) && i < len(format) {
1085         sawSpace = true
1086         i += w
1087         fmtc, w = utf8.DecodeRuneInString(fo
1088     }
1089     if sawSpace {
1090         // There was space in the format, so
1091         // in the input.
1092         inputc := s.getRune()
1093         if inputc == eof {
1094             return
1095         }
1096         if !isSpace(inputc) {
1097             // Space in format but not i
1098             s.errorString("expected spac
1099         }
1100         s.skipSpace(true)
1101         continue
1102     }
1103     inputc := s.mustReadRune()
1104     if fmtc != inputc {
1105         s.UnreadRune()
1106         return -1
1107     }
1108     i += w
1109 }
1110 return
1111 }
1112
1113 // doScanf does the real work when scanning with a format st
1114 // At the moment, it handles only pointers to basic types.
1115 func (s *ss) doScanf(format string, a []interface{}) (numPro
1116     defer errorHandler(&err)
1117     end := len(format) - 1
1118     // We process one item per non-trivial format
1119     for i := 0; i <= end; {
1120         w := s.advance(format[i:])
1121         if w > 0 {
1122             i += w
1123             continue
1124         }
1125         // Either we failed to advance, we have a pe
1126         if format[i] != '%' {
1127             // Can't advance format. Why not?
1128             if w < 0 {
1129                 s.errorString("input does no
1130             }

```

```

1131             // Otherwise at EOF; "too many opera
1132             break
1133         }
1134         i++ // % is one byte
1135
1136         // do we have 20 (width)?
1137         var widPresent bool
1138         s.maxWid, widPresent, i = parsenum(format, i
1139         if !widPresent {
1140             s.maxWid = hugeWid
1141         }
1142         s.fieldLimit = s.limit
1143         if f := s.count + s.maxWid; f < s.fieldLimit
1144             s.fieldLimit = f
1145     }
1146
1147     c, w := utf8.DecodeRuneInString(format[i:])
1148     i += w
1149
1150     if numProcessed >= len(a) { // out of operan
1151         s.errorString("too few operands for
1152         break
1153     }
1154     field := a[numProcessed]
1155
1156     s.scanOne(c, field)
1157     numProcessed++
1158     s.fieldLimit = s.limit
1159 }
1160 if numProcessed < len(a) {
1161     s.errorString("too many operands")
1162 }
1163 return
1164 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/ast/ast.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package ast declares the types used to represent syntax t
6 // packages.
7 //
8 package ast
9
10 import (
11     "go/token"
12     "strings"
13     "unicode"
14     "unicode/utf8"
15 )
16
17 // -----
18 // Interfaces
19 //
20 // There are 3 main classes of nodes: Expressions and type n
21 // statement nodes, and declaration nodes. The node names us
22 // match the corresponding Go spec production names to which
23 // correspond. The node fields correspond to the individual
24 // of the respective productions.
25 //
26 // All nodes contain position information marking the beginn
27 // the corresponding source text segment; it is accessible v
28 // Pos accessor method. Nodes may contain additional positio
29 // for language constructs where comments may be found betwe
30 // of the construct (typically any larger, parenthesized sub
31 // That position information is needed to properly position
32 // when printing the construct.
33
34 // All node types implement the Node interface.
35 type Node interface {
36     Pos() token.Pos // position of first character belong
37     End() token.Pos // position of first character immediat
38 }
39
40 // All expression nodes implement the Expr interface.
41 type Expr interface {
42     Node
43     exprNode()
44 }
```

```

45
46 // All statement nodes implement the Stmt interface.
47 type Stmt interface {
48     Node
49     stmtNode()
50 }
51
52 // All declaration nodes implement the Decl interface.
53 type Decl interface {
54     Node
55     declNode()
56 }
57
58 // -----
59 // Comments
60
61 // A Comment node represents a single //-style or /*-style c
62 type Comment struct {
63     Slash token.Pos // position of "/" starting the comm
64     Text string // comment text (excluding '\n' for
65 }
66
67 func (c *Comment) Pos() token.Pos { return c.Slash }
68 func (c *Comment) End() token.Pos { return token.Pos(int(c.S
69
70 // A CommentGroup represents a sequence of comments
71 // with no other tokens and no empty lines between.
72 //
73 type CommentGroup struct {
74     List []*Comment // len(List) > 0
75 }
76
77 func (g *CommentGroup) Pos() token.Pos { return g.List[0].Po
78 func (g *CommentGroup) End() token.Pos { return g.List[len(g
79
80 func isWhitespace(ch byte) bool { return ch == ' ' || ch ==
81
82 func stripTrailingWhitespace(s string) string {
83     i := len(s)
84     for i > 0 && isWhitespace(s[i-1]) {
85         i--
86     }
87     return s[0:i]
88 }
89
90 // Text returns the text of the comment,
91 // with the comment markers - //, /*, and */ - removed.
92 func (g *CommentGroup) Text() string {
93     if g == nil {
94         return ""

```

```

95     }
96     comments := make([]string, len(g.List))
97     for i, c := range g.List {
98         comments[i] = string(c.Text)
99     }
100
101     lines := make([]string, 0, 10) // most comments are
102     for _, c := range comments {
103         // Remove comment markers.
104         // The parser has given us exactly the comment
105         switch c[1] {
106             case '/':
107                 // -style comment
108                 c = c[2:]
109                 // Remove leading space after //, if
110                 // TODO(gri) This appears to be necessary
111                 // cases (bigNum.RatFromStr)
112                 if len(c) > 0 && c[0] == ' ' {
113                     c = c[1:]
114                 }
115             case '*':
116                 /* -style comment */
117                 c = c[2 : len(c)-2]
118             }
119
120         // Split on newlines.
121         cl := strings.Split(c, "\n")
122
123         // Walk lines, stripping trailing white space
124         for _, l := range cl {
125             lines = append(lines, stripTrailingW
126         }
127     }
128
129     // Remove leading blank lines; convert runs of
130     // interior blank lines to a single blank line.
131     n := 0
132     for _, line := range lines {
133         if line != "" || n > 0 && lines[n-1] != "" {
134             lines[n] = line
135             n++
136         }
137     }
138     lines = lines[0:n]
139
140     // Add final "" entry to get trailing newline from J
141     if n > 0 && lines[n-1] != "" {
142         lines = append(lines, "")
143     }

```

```

144
145         return strings.Join(lines, "\n")
146     }
147
148     // -----
149     // Expressions and types
150
151     // A Field represents a Field declaration list in a struct t
152     // a method list in an interface type, or a parameter/result
153     // in a signature.
154     //
155     type Field struct {
156         Doc      *CommentGroup // associated documentation; o
157         Names   []*Ident      // field/method/parameter name
158         Type    Expr        // field/method/parameter type
159         Tag     *BasicLit    // field tag; or nil
160         Comment *CommentGroup // line comments; or nil
161     }
162
163     func (f *Field) Pos() token.Pos {
164         if len(f.Names) > 0 {
165             return f.Names[0].Pos()
166         }
167         return f.Type.Pos()
168     }
169
170     func (f *Field) End() token.Pos {
171         if f.Tag != nil {
172             return f.Tag.End()
173         }
174         return f.Type.End()
175     }
176
177     // A FieldList represents a list of Fields, enclosed by pare
178     type FieldList struct {
179         Opening token.Pos // position of opening parenthesis
180         List    []*Field // field list; or nil
181         Closing token.Pos // position of closing parenthesis
182     }
183
184     func (f *FieldList) Pos() token.Pos {
185         if f.Opening.IsValid() {
186             return f.Opening
187         }
188         // the list should not be empty in this case;
189         // be conservative and guard against bad ASTs
190         if len(f.List) > 0 {
191             return f.List[0].Pos()
192         }

```

```

193         return token.NoPos
194     }
195
196     func (f *FieldList) End() token.Pos {
197         if f.Closing.IsValid() {
198             return f.Closing + 1
199         }
200         // the list should not be empty in this case;
201         // be conservative and guard against bad ASTs
202         if n := len(f.List); n > 0 {
203             return f.List[n-1].End()
204         }
205         return token.NoPos
206     }
207
208     // NumFields returns the number of (named and anonymous fiel
209     func (f *FieldList) NumFields() int {
210         n := 0
211         if f != nil {
212             for _, g := range f.List {
213                 m := len(g.Names)
214                 if m == 0 {
215                     m = 1 // anonymous field
216                 }
217                 n += m
218             }
219         }
220         return n
221     }
222
223     // An expression is represented by a tree consisting of one
224     // or more of the following concrete expression nodes.
225     //
226     type (
227         // A BadExpr node is a placeholder for expressions c
228         // syntax errors for which no correct expression nod
229         // created.
230         //
231         BadExpr struct {
232             From, To token.Pos // position range of bad
233         }
234
235         // An Ident node represents an identifier.
236         Ident struct {
237             NamePos token.Pos // identifier position
238             Name     string   // identifier name
239             Obj      *Object // denoted object; or nil
240         }
241
242         // An Ellipsis node stands for the "..." type in a

```

```

243 // parameter list or the "... " length in an array ty
244 //
245 Ellipsis struct {
246     Ellipsis token.Pos // position of "... "
247     Elt      Expr      // ellipsis element type
248 }
249
250 // A BasicLit node represents a literal of basic typ
251 BasicLit struct {
252     ValuePos token.Pos // literal position
253     Kind     token.Token // token.INT, token.FLC
254     Value    string     // literal string; e.g.
255 }
256
257 // A FuncLit node represents a function literal.
258 FuncLit struct {
259     Type *FuncType // function type
260     Body *BlockStmt // function body
261 }
262
263 // A CompositeLit node represents a composite litera
264 CompositeLit struct {
265     Type Expr // literal type; or nil
266     Lbrace token.Pos // position of "{"
267     Elts  []Expr // list of composite elemen
268     Rbrace token.Pos // position of "}"
269 }
270
271 // A ParenExpr node represents a parenthesized expre
272 ParenExpr struct {
273     Lparen token.Pos // position of "("
274     X      Expr      // parenthesized expression
275     Rparen token.Pos // position of ")"
276 }
277
278 // A SelectorExpr node represents an expression foll
279 SelectorExpr struct {
280     X Expr // expression
281     Sel *Ident // field selector
282 }
283
284 // An IndexExpr node represents an expression follow
285 IndexExpr struct {
286     X Expr // expression
287     Lbrack token.Pos // position of "["
288     Index Expr // index expression
289     Rbrack token.Pos // position of "]"
290 }
291

```

```

292 // An SliceExpr node represents an expression follow
293 SliceExpr struct {
294     X      Expr      // expression
295     Lbrack token.Pos // position of "["
296     Low   Expr      // begin of slice range; or
297     High  Expr      // end of slice range; or n
298     Rbrack token.Pos // position of "]"
299 }
300
301 // A TypeAssertExpr node represents an expression fo
302 // type assertion.
303 //
304 TypeAssertExpr struct {
305     X      Expr // expression
306     Type  Expr // asserted type; nil means type s
307 }
308
309 // A CallExpr node represents an expression followed
310 CallExpr struct {
311     Fun      Expr      // function expression
312     Lparen   token.Pos // position of "("
313     Args    []Expr     // function arguments; or
314     Ellipsis token.Pos // position of "...", if
315     Rparen   token.Pos // position of ")"
316 }
317
318 // A StarExpr node represents an expression of the f
319 // Semantically it could be a unary "*" expression,
320 //
321 StarExpr struct {
322     Star token.Pos // position of "*"
323     X     Expr      // operand
324 }
325
326 // A UnaryExpr node represents a unary expression.
327 // Unary "*" expressions are represented via StarExp
328 //
329 UnaryExpr struct {
330     OpPos token.Pos // position of Op
331     Op    token.Token // operator
332     X     Expr      // operand
333 }
334
335 // A BinaryExpr node represents a binary expression.
336 BinaryExpr struct {
337     X      Expr      // left operand
338     OpPos token.Pos // position of Op
339     Op    token.Token // operator
340     Y     Expr      // right operand

```

```

341     }
342
343     // A KeyValueExpr node represents (key : value) pair
344     // in composite literals.
345     //
346     KeyValueExpr struct {
347         Key    Expr
348         Colon token.Pos // position of ":"
349         Value Expr
350     }
351 )
352
353 // The direction of a channel type is indicated by one
354 // of the following constants.
355 //
356 type ChanDir int
357
358 const (
359     SEND ChanDir = 1 << iota
360     RECV
361 )
362
363 // A type is represented by a tree consisting of one
364 // or more of the following type-specific expression
365 // nodes.
366 //
367 type (
368     // An ArrayType node represents an array or slice ty
369     ArrayType struct {
370         Lbrack token.Pos // position of "["
371         Len    Expr      // Ellipsis node for [...]T
372         Elt    Expr      // element type
373     }
374
375     // A StructType node represents a struct type.
376     StructType struct {
377         Struct    token.Pos // position of "struct"
378         Fields    *FieldList // list of field decla
379         Incomplete bool      // true if (source) fi
380     }
381
382     // Pointer types are represented via StarExpr nodes.
383
384     // A FuncType node represents a function type.
385     FuncType struct {
386         Func    token.Pos // position of "func" key
387         Params  *FieldList // (incoming) parameters;
388         Results *FieldList // (outgoing) results; or
389     }
390

```

```

391 // An InterfaceType node represents an interface typ
392 InterfaceType struct {
393     Interface token.Pos // position of "interf
394     Methods   *FieldList // list of methods
395     Incomplete bool      // true if (source) me
396 }
397
398 // A MapType node represents a map type.
399 MapType struct {
400     Map token.Pos // position of "map" keyword
401     Key  Expr
402     Value Expr
403 }
404
405 // A ChanType node represents a channel type.
406 ChanType struct {
407     Begin token.Pos // position of "chan" keywor
408     Dir   ChanDir   // channel direction
409     Value Expr      // value type
410 }
411 )
412
413 // Pos and End implementations for expression/type nodes.
414 //
415 func (x *BadExpr) Pos() token.Pos { return x.From }
416 func (x *Ident) Pos() token.Pos   { return x.NamePos }
417 func (x *Ellipsis) Pos() token.Pos { return x.Ellipsis }
418 func (x *BasicLit) Pos() token.Pos { return x.ValuePos }
419 func (x *FuncLit) Pos() token.Pos  { return x.Type.Pos() }
420 func (x *CompositeLit) Pos() token.Pos {
421     if x.Type != nil {
422         return x.Type.Pos()
423     }
424     return x.Lbrace
425 }
426 func (x *ParenExpr) Pos() token.Pos { return x.Lparen }
427 func (x *SelectorExpr) Pos() token.Pos { return x.X.Pos() }
428 func (x *IndexExpr) Pos() token.Pos { return x.X.Pos() }
429 func (x *SliceExpr) Pos() token.Pos { return x.X.Pos() }
430 func (x *TypeAssertExpr) Pos() token.Pos { return x.X.Pos() }
431 func (x *CallExpr) Pos() token.Pos { return x.Fun.Pos() }
432 func (x *StarExpr) Pos() token.Pos { return x.Star }
433 func (x *UnaryExpr) Pos() token.Pos { return x.OpPos }
434 func (x *BinaryExpr) Pos() token.Pos { return x.X.Pos() }
435 func (x *KeyValueExpr) Pos() token.Pos { return x.Key.Pos() }
436 func (x *ArrayType) Pos() token.Pos { return x.Lbrack }
437 func (x *StructType) Pos() token.Pos { return x.Struct }
438 func (x *FuncType) Pos() token.Pos { return x.Func }
439 func (x *InterfaceType) Pos() token.Pos { return x.Interfac

```

```

440 func (x *MapType) Pos() token.Pos      { return x.Map }
441 func (x *ChanType) Pos() token.Pos    { return x.Begin }
442
443 func (x *BadExpr) End() token.Pos { return x.To }
444 func (x *Ident) End() token.Pos  { return token.Pos(int(x.N
445 func (x *Ellipsis) End() token.Pos {
446     if x.Elt != nil {
447         return x.Elt.End()
448     }
449     return x.Ellipsis + 3 // len("...")
450 }
451 func (x *BasicLit) End() token.Pos    { return token.Pos(in
452 func (x *FuncLit) End() token.Pos     { return x.Body.End()
453 func (x *CompositeLit) End() token.Pos { return x.Rbrace + 1
454 func (x *ParenExpr) End() token.Pos   { return x.Rparen + 1
455 func (x *SelectorExpr) End() token.Pos { return x.Sel.End()
456 func (x *IndexExpr) End() token.Pos   { return x.Rbrack + 1
457 func (x *SliceExpr) End() token.Pos   { return x.Rbrack + 1
458 func (x *TypeAssertExpr) End() token.Pos {
459     if x.Type != nil {
460         return x.Type.End()
461     }
462     return x.X.End()
463 }
464 func (x *CallExpr) End() token.Pos    { return x.Rparen + 1
465 func (x *StarExpr) End() token.Pos    { return x.X.End() }
466 func (x *UnaryExpr) End() token.Pos   { return x.X.End() }
467 func (x *BinaryExpr) End() token.Pos  { return x.Y.End() }
468 func (x *KeyValueExpr) End() token.Pos { return x.Value.End(
469 func (x *ArrayType) End() token.Pos   { return x.Elt.End()
470 func (x *StructType) End() token.Pos  { return x.Fields.End
471 func (x *FuncType) End() token.Pos {
472     if x.Results != nil {
473         return x.Results.End()
474     }
475     return x.Params.End()
476 }
477 func (x *InterfaceType) End() token.Pos { return x.Methods.E
478 func (x *MapType) End() token.Pos      { return x.Value.End
479 func (x *ChanType) End() token.Pos     { return x.Value.End
480
481 // exprNode() ensures that only expression/type nodes can be
482 // assigned to an ExprNode.
483 //
484 func (*BadExpr) exprNode()    {}
485 func (*Ident) exprNode()     {}
486 func (*Ellipsis) exprNode()  {}
487 func (*BasicLit) exprNode()  {}
488 func (*FuncLit) exprNode()   {}

```

```

489 func (*CompositeLit) exprNode() {}
490 func (*ParenExpr) exprNode() {}
491 func (*SelectorExpr) exprNode() {}
492 func (*IndexExpr) exprNode() {}
493 func (*SliceExpr) exprNode() {}
494 func (*TypeAssertExpr) exprNode() {}
495 func (*CallExpr) exprNode() {}
496 func (*StarExpr) exprNode() {}
497 func (*UnaryExpr) exprNode() {}
498 func (*BinaryExpr) exprNode() {}
499 func (*KeyValueExpr) exprNode() {}
500
501 func (*ArrayType) exprNode() {}
502 func (*StructType) exprNode() {}
503 func (*FuncType) exprNode() {}
504 func (*InterfaceType) exprNode() {}
505 func (*MapType) exprNode() {}
506 func (*ChanType) exprNode() {}
507
508 // -----
509 // Convenience functions for Idents
510
511 var noPos token.Pos
512
513 // NewIdent creates a new Ident without position.
514 // Useful for ASTs generated by code other than the Go parse
515 //
516 func NewIdent(name string) *Ident { return &Ident{noPos, name} }
517
518 // IsExported returns whether name is an exported Go symbol
519 // (i.e., whether it begins with an uppercase letter).
520 //
521 func IsExported(name string) bool {
522     ch, _ := utf8.DecodeRuneInString(name)
523     return unicode.IsUpper(ch)
524 }
525
526 // IsExported returns whether id is an exported Go symbol
527 // (i.e., whether it begins with an uppercase letter).
528 //
529 func (id *Ident) IsExported() bool { return IsExported(id.Name) }
530
531 func (id *Ident) String() string {
532     if id != nil {
533         return id.Name
534     }
535     return "<nil>"
536 }
537
538 // -----

```

```

539 // Statements
540
541 // A statement is represented by a tree consisting of one
542 // or more of the following concrete statement nodes.
543 //
544 type (
545     // A BadStmt node is a placeholder for statements co
546     // syntax errors for which no correct statement node
547     // created.
548     //
549     BadStmt struct {
550         From, To token.Pos // position range of bad
551     }
552
553     // A DeclStmt node represents a declaration in a sta
554     DeclStmt struct {
555         Decl Decl
556     }
557
558     // An EmptyStmt node represents an empty statement.
559     // The "position" of the empty statement is the posi
560     // of the immediately preceding semicolon.
561     //
562     EmptyStmt struct {
563         Semicolon token.Pos // position of preceding
564     }
565
566     // A LabeledStmt node represents a labeled statement
567     LabeledStmt struct {
568         Label *Ident
569         Colon token.Pos // position of ":"
570         Stmt Stmt
571     }
572
573     // An ExprStmt node represents a (stand-alone) expre
574     // in a statement list.
575     //
576     ExprStmt struct {
577         X Expr // expression
578     }
579
580     // A SendStmt node represents a send statement.
581     SendStmt struct {
582         Chan Expr
583         Arrow token.Pos // position of "<-"
584         Value Expr
585     }
586
587     // An IncDecStmt node represents an increment or dec

```

```

588     IncDecStmt struct {
589         X      Expr
590         TokPos token.Pos // position of Tok
591         Tok    token.Token // INC or DEC
592     }
593
594     // An AssignStmt node represents an assignment or
595     // a short variable declaration.
596     //
597     AssignStmt struct {
598         Lhs    []Expr
599         TokPos token.Pos // position of Tok
600         Tok    token.Token // assignment token, DEFI
601         Rhs    []Expr
602     }
603
604     // A GoStmt node represents a go statement.
605     GoStmt struct {
606         Go    token.Pos // position of "go" keyword
607         Call *CallExpr
608     }
609
610     // A DeferStmt node represents a defer statement.
611     DeferStmt struct {
612         Defer token.Pos // position of "defer" keywo
613         Call  *CallExpr
614     }
615
616     // A ReturnStmt node represents a return statement.
617     ReturnStmt struct {
618         Return token.Pos // position of "return" ke
619         Results []Expr    // result expressions; or
620     }
621
622     // A BranchStmt node represents a break, continue, g
623     // or fallthrough statement.
624     //
625     BranchStmt struct {
626         TokPos token.Pos // position of Tok
627         Tok    token.Token // keyword token (BREAK,
628         Label  *Ident     // label name; or nil
629     }
630
631     // A BlockStmt node represents a braced statement li
632     BlockStmt struct {
633         Lbrace token.Pos // position of "{"
634         List   []Stmt
635         Rbrace token.Pos // position of "}"
636     }

```

```

637
638 // An IfStmt node represents an if statement.
639 IfStmt struct {
640     If token.Pos // position of "if" keyword
641     Init Stmt    // initialization statement;
642     Cond Expr    // condition
643     Body *BlockStmt
644     Else Stmt // else branch; or nil
645 }
646
647 // A CaseClause represents a case of an expression o
648 CaseClause struct {
649     Case token.Pos // position of "case" or "de
650     List []Expr    // list of expressions or ty
651     Colon token.Pos // position of ":"
652     Body []Stmt    // statement list; or nil
653 }
654
655 // A SwitchStmt node represents an expression switch
656 SwitchStmt struct {
657     Switch token.Pos // position of "switch" ke
658     Init Stmt        // initialization statemen
659     Tag Expr         // tag expression; or nil
660     Body *BlockStmt // CaseClauses only
661 }
662
663 // An TypeSwitchStmt node represents a type switch s
664 TypeSwitchStmt struct {
665     Switch token.Pos // position of "switch" ke
666     Init Stmt        // initialization statemen
667     Assign Stmt      // x := y.(type) or y.(typ
668     Body *BlockStmt // CaseClauses only
669 }
670
671 // A CommClause node represents a case of a select s
672 CommClause struct {
673     Case token.Pos // position of "case" or "de
674     Comm Stmt      // send or receive statement
675     Colon token.Pos // position of ":"
676     Body []Stmt    // statement list; or nil
677 }
678
679 // An SelectStmt node represents a select statement.
680 SelectStmt struct {
681     Select token.Pos // position of "select" ke
682     Body *BlockStmt // CommClauses only
683 }
684
685 // A ForStmt represents a for statement.
686 ForStmt struct {

```

```

687         For token.Pos // position of "for" keyword
688         Init Stmt      // initialization statement;
689         Cond Expr      // condition; or nil
690         Post Stmt     // post iteration statement;
691         Body *BlockStmt
692     }
693
694     // A RangeStmt represents a for statement with a ran
695     RangeStmt struct {
696         For          token.Pos // position of "for"
697         Key, Value Expr      // Value may be nil
698         TokPos       token.Pos // position of Tok
699         Tok          token.Token // ASSIGN, DEFINE
700         X           Expr      // value to range ove
701         Body        *BlockStmt
702     }
703 )
704
705 // Pos and End implementations for statement nodes.
706 //
707 func (s *BadStmt) Pos() token.Pos      { return s.From }
708 func (s *DeclStmt) Pos() token.Pos     { return s.Decl.Pos }
709 func (s *EmptyStmt) Pos() token.Pos    { return s.Semicolo
710 func (s *LabeledStmt) Pos() token.Pos  { return s.Label.Po
711 func (s *ExprStmt) Pos() token.Pos     { return s.X.Pos()
712 func (s *SendStmt) Pos() token.Pos     { return s.Chan.Pos
713 func (s *IncDecStmt) Pos() token.Pos   { return s.X.Pos()
714 func (s *AssignStmt) Pos() token.Pos   { return s.Lhs[0].P
715 func (s *GoStmt) Pos() token.Pos       { return s.Go }
716 func (s *DeferStmt) Pos() token.Pos    { return s.Defer }
717 func (s *ReturnStmt) Pos() token.Pos   { return s.Return }
718 func (s *BranchStmt) Pos() token.Pos   { return s.TokPos }
719 func (s *BlockStmt) Pos() token.Pos    { return s.Lbrace }
720 func (s *IfStmt) Pos() token.Pos       { return s.If }
721 func (s *CaseClause) Pos() token.Pos   { return s.Case }
722 func (s *SwitchStmt) Pos() token.Pos   { return s.Switch }
723 func (s *TypeSwitchStmt) Pos() token.Pos { return s.Switch }
724 func (s *CommClause) Pos() token.Pos   { return s.Case }
725 func (s *SelectStmt) Pos() token.Pos   { return s.Select }
726 func (s *ForStmt) Pos() token.Pos      { return s.For }
727 func (s *RangeStmt) Pos() token.Pos    { return s.For }
728
729 func (s *BadStmt) End() token.Pos      { return s.To }
730 func (s *DeclStmt) End() token.Pos     { return s.Decl.End() }
731 func (s *EmptyStmt) End() token.Pos   {
732     return s.Semicolon + 1 /* len(";") */
733 }
734 func (s *LabeledStmt) End() token.Pos  { return s.Stmt.End()
735 func (s *ExprStmt) End() token.Pos     { return s.X.End() }

```

```

736 func (s *SendStmt) End() token.Pos { return s.Value.End()
737 func (s *IncDecStmt) End() token.Pos {
738     return s.TokPos + 2 /* len("++") */
739 }
740 func (s *AssignStmt) End() token.Pos { return s.Rhs[len(s.Rh
741 func (s *GoStmt) End() token.Pos { return s.Call.End() }
742 func (s *DeferStmt) End() token.Pos { return s.Call.End() }
743 func (s *ReturnStmt) End() token.Pos {
744     if n := len(s.Results); n > 0 {
745         return s.Results[n-1].End()
746     }
747     return s.Return + 6 // len("return")
748 }
749 func (s *BranchStmt) End() token.Pos {
750     if s.Label != nil {
751         return s.Label.End()
752     }
753     return token.Pos(int(s.TokPos) + len(s.Tok.String()))
754 }
755 func (s *BlockStmt) End() token.Pos { return s.Rbrace + 1 }
756 func (s *IfStmt) End() token.Pos {
757     if s.Else != nil {
758         return s.Else.End()
759     }
760     return s.Body.End()
761 }
762 func (s *CaseClause) End() token.Pos {
763     if n := len(s.Body); n > 0 {
764         return s.Body[n-1].End()
765     }
766     return s.Colon + 1
767 }
768 func (s *SwitchStmt) End() token.Pos { return s.Body.End
769 func (s *TypeSwitchStmt) End() token.Pos { return s.Body.End
770 func (s *CommClause) End() token.Pos {
771     if n := len(s.Body); n > 0 {
772         return s.Body[n-1].End()
773     }
774     return s.Colon + 1
775 }
776 func (s *SelectStmt) End() token.Pos { return s.Body.End() }
777 func (s *ForStmt) End() token.Pos { return s.Body.End() }
778 func (s *RangeStmt) End() token.Pos { return s.Body.End() }
779
780 // stmtNode() ensures that only statement nodes can be
781 // assigned to a StmtNode.
782 //
783 func (*BadStmt) stmtNode() {}
784 func (*DeclStmt) stmtNode() {}

```

```

785 func (*EmptyStmt) stmtNode()      {}
786 func (*LabeledStmt) stmtNode()    {}
787 func (*ExprStmt) stmtNode()       {}
788 func (*SendStmt) stmtNode()       {}
789 func (*IncDecStmt) stmtNode()     {}
790 func (*AssignStmt) stmtNode()     {}
791 func (*GoStmt) stmtNode()         {}
792 func (*DeferStmt) stmtNode()      {}
793 func (*ReturnStmt) stmtNode()     {}
794 func (*BranchStmt) stmtNode()    {}
795 func (*BlockStmt) stmtNode()      {}
796 func (*IfStmt) stmtNode()        {}
797 func (*CaseClause) stmtNode()    {}
798 func (*SwitchStmt) stmtNode()    {}
799 func (*TypeSwitchStmt) stmtNode() {}
800 func (*CommClause) stmtNode()    {}
801 func (*SelectStmt) stmtNode()    {}
802 func (*ForStmt) stmtNode()       {}
803 func (*RangeStmt) stmtNode()     {}
804
805 // -----
806 // Declarations
807
808 // A Spec node represents a single (non-parenthesized) impor
809 // constant, type, or variable declaration.
810 //
811 type (
812     // The Spec type stands for any of *ImportSpec, *Val
813     Spec interface {
814         Node
815         specNode()
816     }
817
818     // An ImportSpec node represents a single package im
819     ImportSpec struct {
820         Doc      *CommentGroup // associated document
821         Name     *Ident         // local package name
822         Path     *BasicLit     // import path
823         Comment  *CommentGroup // line comments; or n
824         EndPos   token.Pos     // end of spec (overri
825     }
826
827     // A ValueSpec node represents a constant or variabl
828     // (ConstSpec or VarSpec production).
829     //
830     ValueSpec struct {
831         Doc      *CommentGroup // associated document
832         Names    []*Ident      // value names (len(Na
833         Type    Expr         // value type; or nil
834         Values  []Expr       // initial values; or

```

```

835         Comment *CommentGroup // line comments; or n
836     }
837
838     // A TypeSpec node represents a type declaration (Ty
839     TypeSpec struct {
840         Doc      *CommentGroup // associated document
841         Name     *Ident        // type name
842         Type     Expr          // *Ident, *ParenExpr,
843         Comment  *CommentGroup // line comments; or n
844     }
845 )
846
847 // Pos and End implementations for spec nodes.
848 //
849 func (s *ImportSpec) Pos() token.Pos {
850     if s.Name != nil {
851         return s.Name.Pos()
852     }
853     return s.Path.Pos()
854 }
855 func (s *ValueSpec) Pos() token.Pos { return s.Names[0].Pos()
856 func (s *TypeSpec) Pos() token.Pos { return s.Name.Pos() }
857
858 func (s *ImportSpec) End() token.Pos {
859     if s.EndPos != 0 {
860         return s.EndPos
861     }
862     return s.Path.End()
863 }
864
865 func (s *ValueSpec) End() token.Pos {
866     if n := len(s.Values); n > 0 {
867         return s.Values[n-1].End()
868     }
869     if s.Type != nil {
870         return s.Type.End()
871     }
872     return s.Names[len(s.Names)-1].End()
873 }
874 func (s *TypeSpec) End() token.Pos { return s.Type.End() }
875
876 // specNode() ensures that only spec nodes can be
877 // assigned to a Spec.
878 //
879 func (*ImportSpec) specNode() {}
880 func (*ValueSpec) specNode() {}
881 func (*TypeSpec) specNode() {}
882
883 // A declaration is represented by one of the following decl

```

```

884 //
885 type (
886     // A BadDecl node is a placeholder for declarations
887     // syntax errors for which no correct declaration no
888     // created.
889     //
890     BadDecl struct {
891         From, To token.Pos // position range of bad
892     }
893
894     // A GenDecl node (generic declaration node) represe
895     // constant, type or variable declaration. A valid L
896     // (Lparen.Line > 0) indicates a parenthesized decla
897     //
898     // Relationship between Tok value and Specs element
899     //
900     //     token.IMPORT  *ImportSpec
901     //     token.CONST   *ValueSpec
902     //     token.TYPE    *TypeSpec
903     //     token.VAR     *ValueSpec
904     //
905     GenDecl struct {
906         Doc      *CommentGroup // associated documenta
907         TokPos   token.Pos      // position of Tok
908         Tok      token.Token    // IMPORT, CONST, TYPE,
909         Lparen   token.Pos      // position of '(', if
910         Specs    []Spec
911         Rparen   token.Pos      // position of ')', if any
912     }
913
914     // A FuncDecl node represents a function declaration
915     FuncDecl struct {
916         Doc      *CommentGroup // associated documentati
917         Recv    *FieldList     // receiver (methods); or
918         Name    *Ident         // function/method name
919         Type    *FuncType      // position of Func keywo
920         Body    *BlockStmt     // function body; or nil
921     }
922 )
923
924 // Pos and End implementations for declaration nodes.
925 //
926 func (d *BadDecl) Pos() token.Pos { return d.From }
927 func (d *GenDecl) Pos() token.Pos { return d.TokPos }
928 func (d *FuncDecl) Pos() token.Pos { return d.Type.Pos() }
929
930 func (d *BadDecl) End() token.Pos { return d.To }
931 func (d *GenDecl) End() token.Pos {
932     if d.Rparen.IsValid() {

```

```

933         return d.Rparen + 1
934     }
935     return d.Specs[0].End()
936 }
937 func (d *FuncDecl) End() token.Pos {
938     if d.Body != nil {
939         return d.Body.End()
940     }
941     return d.Type.End()
942 }
943
944 // declNode() ensures that only declaration nodes can be
945 // assigned to a DeclNode.
946 //
947 func (*BadDecl) declNode() {}
948 func (*GenDecl) declNode() {}
949 func (*FuncDecl) declNode() {}
950
951 // -----
952 // Files and packages
953
954 // A File node represents a Go source file.
955 //
956 // The Comments list contains all comments in the source file
957 // appearance, including the comments that are pointed to from
958 // via Doc and Comment fields.
959 //
960 type File struct {
961     Doc          *CommentGroup // associated documentation
962     Package      token.Pos      // position of "package"
963     Name         *Ident         // package name
964     Decls        []Decl         // top-level declarations
965     Scope        *Scope         // package scope (this file)
966     Imports      []*ImportSpec  // imports in this file
967     Unresolved   []*Ident       // unresolved identifiers
968     Comments     []*CommentGroup // list of all comments in this file
969 }
970
971 func (f *File) Pos() token.Pos { return f.Package }
972 func (f *File) End() token.Pos {
973     if n := len(f.Decls); n > 0 {
974         return f.Decls[n-1].End()
975     }
976     return f.Name.End()
977 }
978
979 // A Package node represents a set of source files
980 // collectively building a Go package.
981 //
982 type Package struct {

```

```
983         Name    string           // package name
984         Scope   *Scope           // package scope across a
985         Imports map[string]*Object // map of package id -> p
986         Files   map[string]*File   // Go source files by fil
987     }
988
989     func (p *Package) Pos() token.Pos { return token.NoPos }
990     func (p *Package) End() token.Pos { return token.NoPos }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/ast/filter.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package ast
6
7 import (
8     "go/token"
9     "sort"
10 )
11
12 // -----
13 // Export filtering
14
15 // exportFilter is a special filter function to extract exported
16 func exportFilter(name string) bool {
17     return IsExported(name)
18 }
19
20 // FileExports trims the AST for a Go source file in place s
21 // only exported nodes remain: all top-level identifiers whi
22 // and their associated information (such as type, initial v
23 // body) are removed. Non-exported fields and methods of exp
24 // stripped. The File.Comments list is not changed.
25 //
26 // FileExports returns true if there are exported declaratio
27 // it returns false otherwise.
28 //
29 func FileExports(src *File) bool {
30     return filterFile(src, exportFilter, true)
31 }
32
33 // PackageExports trims the AST for a Go package in place su
34 // only exported nodes remain. The pkg.Files list is not cha
35 // file names and top-level package comments don't get lost.
36 //
37 // PackageExports returns true if there are exported declara
38 // it returns false otherwise.
39 //
40 func PackageExports(pkg *Package) bool {
41     return filterPackage(pkg, exportFilter, true)
42 }
43
44 // -----
```

```

45 // General filtering
46
47 type Filter func(string) bool
48
49 func filterIdentList(list []*Ident, f Filter) []*Ident {
50     j := 0
51     for _, x := range list {
52         if f(x.Name) {
53             list[j] = x
54             j++
55         }
56     }
57     return list[0:j]
58 }
59
60 // fieldName assumes that x is the type of an anonymous fiel
61 // returns the corresponding field name. If x is not an acce
62 // anonymous field, the result is nil.
63 //
64 func fieldName(x Expr) *Ident {
65     switch t := x.(type) {
66     case *Ident:
67         return t
68     case *SelectorExpr:
69         if _, ok := t.X.(*Ident); ok {
70             return t.Sel
71         }
72     case *StarExpr:
73         return fieldName(t.X)
74     }
75     return nil
76 }
77
78 func filterFieldList(fields *FieldList, filter Filter, expor
79     if fields == nil {
80         return false
81     }
82     list := fields.List
83     j := 0
84     for _, f := range list {
85         keepField := false
86         if len(f.Names) == 0 {
87             // anonymous field
88             name := fieldName(f.Type)
89             keepField = name != nil && filter(na
90         } else {
91             n := len(f.Names)
92             f.Names = filterIdentList(f.Names, f
93             if len(f.Names) < n {
94                 removedFields = true

```

```

95         }
96         keepField = len(f.Names) > 0
97     }
98     if keepField {
99         if export {
100             filterType(f.Type, filter, e
101         }
102         list[j] = f
103         j++
104     }
105 }
106 if j < len(list) {
107     removedFields = true
108 }
109 fields.List = list[0:j]
110 return
111 }
112
113 func filterParamList(fields *FieldList, filter Filter, export
114     if fields == nil {
115         return false
116     }
117     var b bool
118     for _, f := range fields.List {
119         if filterType(f.Type, filter, export) {
120             b = true
121         }
122     }
123     return b
124 }
125
126 func filterType(typ Expr, f Filter, export bool) bool {
127     switch t := typ.(type) {
128     case *Ident:
129         return f(t.Name)
130     case *ParenExpr:
131         return filterType(t.X, f, export)
132     case *ArrayType:
133         return filterType(t.Elt, f, export)
134     case *StructType:
135         if filterFieldList(t.Fields, f, export) {
136             t.Incomplete = true
137         }
138         return len(t.Fields.List) > 0
139     case *FuncType:
140         b1 := filterParamList(t.Params, f, export)
141         b2 := filterParamList(t.Results, f, export)
142         return b1 || b2
143     case *InterfaceType:

```

```

144         if filterFieldList(t.Methods, f, export) {
145             t.Incomplete = true
146         }
147         return len(t.Methods.List) > 0
148     case *MapType:
149         b1 := filterType(t.Key, f, export)
150         b2 := filterType(t.Value, f, export)
151         return b1 || b2
152     case *ChanType:
153         return filterType(t.Value, f, export)
154     }
155     return false
156 }
157
158 func filterSpec(spec Spec, f Filter, export bool) bool {
159     switch s := spec.(type) {
160     case *ValueSpec:
161         s.Names = filterIdentList(s.Names, f)
162         if len(s.Names) > 0 {
163             if export {
164                 filterType(s.Type, f, export)
165             }
166             return true
167         }
168     case *TypeSpec:
169         if f(s.Name.Name) {
170             if export {
171                 filterType(s.Type, f, export)
172             }
173             return true
174         }
175         if !export {
176             // For general filtering (not just e
177             // filter type even if name is not f
178             // out.
179             // If the type contains filtered ele
180             // keep the declaration.
181             return filterType(s.Type, f, export)
182         }
183     }
184     return false
185 }
186
187 func filterSpecList(list []Spec, f Filter, export bool) []Sp
188     j := 0
189     for _, s := range list {
190         if filterSpec(s, f, export) {
191             list[j] = s
192             j++

```

```

193         }
194     }
195     return list[0:j]
196 }
197
198 // FilterDecl trims the AST for a Go declaration in place by
199 // all names (including struct field and interface method na
200 // not from parameter lists) that don't pass through the fil
201 //
202 // FilterDecl returns true if there are any declared names l
203 // filtering; it returns false otherwise.
204 //
205 func FilterDecl(decl Decl, f Filter) bool {
206     return filterDecl(decl, f, false)
207 }
208
209 func filterDecl(decl Decl, f Filter, export bool) bool {
210     switch d := decl.(type) {
211     case *GenDecl:
212         d.Specs = filterSpecList(d.Specs, f, export)
213         return len(d.Specs) > 0
214     case *FuncDecl:
215         return f(d.Name.Name)
216     }
217     return false
218 }
219
220 // FilterFile trims the AST for a Go file in place by removi
221 // names from top-level declarations (including struct field
222 // interface method names, but not from parameter lists) tha
223 // pass through the filter f. If the declaration is empty af
224 // the declaration is removed from the AST. The File.Comment
225 // is not changed.
226 //
227 // FilterFile returns true if there are any top-level declar
228 // left after filtering; it returns false otherwise.
229 //
230 func FilterFile(src *File, f Filter) bool {
231     return filterFile(src, f, false)
232 }
233
234 func filterFile(src *File, f Filter, export bool) bool {
235     j := 0
236     for _, d := range src.Decls {
237         if filterDecl(d, f, export) {
238             src.Decls[j] = d
239             j++
240         }
241     }
242     src.Decls = src.Decls[0:j]

```

```

243         return j > 0
244     }
245
246 // FilterPackage trims the AST for a Go package in place by
247 // all names from top-level declarations (including struct f
248 // interface method names, but not from parameter lists) tha
249 // pass through the filter f. If the declaration is empty af
250 // the declaration is removed from the AST. The pkg.Files li
251 // changed, so that file names and top-level package comment
252 // lost.
253 //
254 // FilterPackage returns true if there are any top-level dec
255 // left after filtering; it returns false otherwise.
256 //
257 func FilterPackage(pkg *Package, f Filter) bool {
258     return filterPackage(pkg, f, false)
259 }
260
261 func filterPackage(pkg *Package, f Filter, export bool) bool
262     hasDecls := false
263     for _, src := range pkg.Files {
264         if filterFile(src, f, export) {
265             hasDecls = true
266         }
267     }
268     return hasDecls
269 }
270
271 // -----
272 // Merging of package files
273
274 // The MergeMode flags control the behavior of MergePackageF
275 type MergeMode uint
276
277 const (
278     // If set, duplicate function declarations are exclu
279     FilterFuncDuplicates MergeMode = 1 << iota
280     // If set, comments that are not associated with a s
281     // AST node (as Doc or Comment) are excluded.
282     FilterUnassociatedComments
283     // If set, duplicate import declarations are exclude
284     FilterImportDuplicates
285 )
286
287 // separator is an empty //-style comment that is interspers
288 // different comment groups when they are concatenated into
289 //
290 var separator = &Comment{noPos, "//"}
291

```

```

292 // MergePackageFiles creates a file AST by merging the ASTs
293 // files belonging to a package. The mode flags control merg
294 //
295 func MergePackageFiles(pkg *Package, mode MergeMode) *File {
296     // Count the number of package docs, comments and de
297     // all package files. Also, compute sorted list of f
298     // subsequent iterations can always iterate in the s
299     ndocs := 0
300     ncomments := 0
301     ndecls := 0
302     filenames := make([]string, len(pkg.Files))
303     i := 0
304     for filename, f := range pkg.Files {
305         filenames[i] = filename
306         i++
307         if f.Doc != nil {
308             ndocs += len(f.Doc.List) + 1 // +1 f
309         }
310         ncomments += len(f.Comments)
311         ndecls += len(f.Decls)
312     }
313     sort.Strings(filenames)
314
315     // Collect package comments from all package files i
316     // CommentGroup - the collected package documentatio
317     // there should be only one file with a package comm
318     // better to collect extra comments than drop them o
319     var doc *CommentGroup
320     var pos token.Pos
321     if ndocs > 0 {
322         list := make([]*Comment, ndocs-1) // -1: no
323         i := 0
324         for _, filename := range filenames {
325             f := pkg.Files[filename]
326             if f.Doc != nil {
327                 if i > 0 {
328                     // not the first gro
329                     list[i] = separator
330                     i++
331                 }
332                 for _, c := range f.Doc.List
333                     list[i] = c
334                     i++
335                 }
336                 if f.Package > pos {
337                     // Keep the maximum
338                     // position for the
339                     // files.
340                     pos = f.Package

```

```

341         }
342     }
343 }
344     doc = &CommentGroup{list}
345 }
346
347 // Collect declarations from all package files.
348 var decls []Decl
349 if ndecls > 0 {
350     decls = make([]Decl, ndecls)
351     funcs := make(map[string]int) // map of glob
352     i := 0 // current ind
353     n := 0 // number of f
354     for _, filename := range filenames {
355         f := pkg.Files[filename]
356         for _, d := range f.Decls {
357             if mode&FilterFuncDuplicates
358                 // A language entity
359                 // times in differen
360                 // build time declar
361                 // For now, exclude
362                 // functions - keep
363                 //
364                 // TODO(gri): Expand
365                 //     entiti
366                 //     multip
367                 if f, isFun := d.(*F
368                     name := f.Na
369                     if j, exists
370                         // f
371                         if d
372
373
374
375                 } e1
376
377
378                 }
379                 n++
380             } else {
381                 func
382             }
383         }
384     }
385     decls[i] = d
386     i++
387 }
388 }
389
390 // Eliminate nil entries from the decls list

```

```

391         // filtered. We do this using a 2nd pass in
392         // the original declaration order in the sou
393         // would also invalidate the monotonically i
394         // info within a single file).
395         if n > 0 {
396             i = 0
397             for _, d := range decls {
398                 if d != nil {
399                     decls[i] = d
400                     i++
401                 }
402             }
403             decls = decls[0:i]
404         }
405     }
406
407     // Collect import specs from all package files.
408     var imports []*ImportSpec
409     if mode&FilterImportDuplicates != 0 {
410         seen := make(map[string]bool)
411         for _, filename := range filenames {
412             f := pkg.Files[filename]
413             for _, imp := range f.Imports {
414                 if path := imp.Path.Value; !
415                     // TODO: consider ha
416                     // - 2 imports exist
417                     //   have different
418                     //   keep both of th
419                     // - 2 imports exist
420                     // - 2 imports exist
421                     //   different) comm
422                     imports = append(imp
423                     seen[path] = true
424                 }
425             }
426         }
427     } else {
428         for _, f := range pkg.Files {
429             imports = append(imports, f.Imports.
430         }
431     }
432
433     // Collect comments from all package files.
434     var comments []*CommentGroup
435     if mode&FilterUnassociatedComments == 0 {
436         comments = make([]*CommentGroup, ncomments)
437         i := 0
438         for _, f := range pkg.Files {
439             i += copy(comments[i:], f.Comments)

```

```
440         }
441     }
442
443     // TODO(gri) need to compute unresolved identifiers!
444     return &File{doc, pos, NewIdent(pkg.Name), decls, pk
445 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/ast/import.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package ast
6
7 import (
8     "go/token"
9     "sort"
10    "strconv"
11 )
12
13 // SortImports sorts runs of consecutive import lines in imp
14 func SortImports(fset *token.FileSet, f *File) {
15     for _, d := range f.Decls {
16         d, ok := d.(*GenDecl)
17         if !ok || d.Tok != token.IMPORT {
18             // Not an import declaration, so we'
19             // Imports are always first.
20             break
21         }
22
23         if d.Lparen == token.NoPos {
24             // Not a block: sorted by default.
25             continue
26         }
27
28         // Identify and sort runs of specs on succes
29         i := 0
30         for j, s := range d.Specs {
31             if j > i && fset.Position(s.Pos()).L
32                 // j begins a new run. End
33                 sortSpecs(fset, f, d.Specs[i
34                 i = j
35             }
36         }
37         sortSpecs(fset, f, d.Specs[i:])
38     }
39 }
40
41 func importPath(s Spec) string {
42     t, err := strconv.Unquote(s.(*ImportSpec).Path.Value
43     if err == nil {
44         return t
```

```

45     }
46     return ""
47 }
48
49 type posSpan struct {
50     Start token.Pos
51     End   token.Pos
52 }
53
54 func sortSpecs(fset *token.FileSet, f *File, specs []Spec) {
55     // Avoid work if already sorted (also catches < 2 en
56     sorted := true
57     for i, s := range specs {
58         if i > 0 && importPath(specs[i-1]) > importP
59             sorted = false
60             break
61     }
62     if sorted {
63         return
64     }
65
66     // Record positions for specs.
67     pos := make([]posSpan, len(specs))
68     for i, s := range specs {
69         pos[i] = posSpan{s.Pos(), s.End()}
70     }
71
72     // Identify comments in this range.
73     // Any comment from pos[0].Start to the final line c
74     lastLine := fset.Position(pos[len(pos)-1].End).Line
75     cstart := len(f.Comments)
76     cend := len(f.Comments)
77     for i, g := range f.Comments {
78         if g.Pos() < pos[0].Start {
79             continue
80         }
81         if i < cstart {
82             cstart = i
83         }
84         if fset.Position(g.End()).Line > lastLine {
85             cend = i
86             break
87         }
88     }
89     comments := f.Comments[cstart:cend]
90
91     // Assign each comment to the import spec preceding
92     importComment := map[*ImportSpec][]*CommentGroup{}
93     specIndex := 0
94

```

```

95     for _, g := range comments {
96         for specIndex+1 < len(specs) && pos[specIndex] < pos[specIndex+1] {
97             specIndex++
98         }
99         s := specs[specIndex].(*ImportSpec)
100        importComment[s] = append(importComment[s],
101    }
102
103    // Sort the import specs by import path.
104    // Reassign the import paths to have the same position.
105    // Reassign each comment to abut the end of its spec.
106    // Sort the comments by new position.
107    sort.Sort(byImportPath(specs))
108    for i, s := range specs {
109        s := s.(*ImportSpec)
110        if s.Name != nil {
111            s.Name.NamePos = pos[i].Start
112        }
113        s.Path.ValuePos = pos[i].Start
114        s.EndPos = pos[i].End
115        for _, g := range importComment[s] {
116            for _, c := range g.List {
117                c.Slash = pos[i].End
118            }
119        }
120    }
121    sort.Sort(byCommentPos(comments))
122 }
123
124 type byImportPath []Spec // slice of *ImportSpec
125
126 func (x byImportPath) Len() int           { return len(x) }
127 func (x byImportPath) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }
128 func (x byImportPath) Less(i, j int) bool { return importPathLess(x[i], x[j]) }
129
130 type byCommentPos []*CommentGroup
131
132 func (x byCommentPos) Len() int           { return len(x) }
133 func (x byCommentPos) Swap(i, j int)      { x[i], x[j] = x[j], x[i] }
134 func (x byCommentPos) Less(i, j int) bool { return x[i].Pos < x[j].Pos }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/ast/print.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file contains printing support for ASTs.
6
7 package ast
8
9 import (
10     "fmt"
11     "go/token"
12     "io"
13     "os"
14     "reflect"
15 )
16
17 // A FieldFilter may be provided to Fprint to control the ou
18 type FieldFilter func(name string, value reflect.Value) bool
19
20 // NotNilFilter returns true for field values that are not n
21 // it returns false otherwise.
22 func NotNilFilter(_ string, v reflect.Value) bool {
23     switch v.Kind() {
24     case reflect.Chan, reflect.Func, reflect.Interface,
25         return !v.IsNil()
26     }
27     return true
28 }
29
30 // Fprint prints the (sub-)tree starting at AST node x to w.
31 // If fset != nil, position information is interpreted relat
32 // to that file set. Otherwise positions are printed as inte
33 // values (file set specific offsets).
34 //
35 // A non-nil FieldFilter f may be provided to control the ou
36 // struct fields for which f(fieldname, fieldvalue) is true
37 // are printed; all others are filtered from the output.
38 //
39 func Fprint(w io.Writer, fset *token.FileSet, x interface{}),
40     // setup printer
41     p := printer{
42         output: w,
43         fset:    fset,
44         filter:  f,
```

```

45         ptrmap: make(map[interface{}]int),
46         last:  '\n', // force printing of line numb
47     }
48
49     // install error handler
50     defer func() {
51         if e := recover(); e != nil {
52             err = e.(localError).err // re-panic
53         }
54     }()
55
56     // print x
57     if x == nil {
58         p.printf("nil\n")
59         return
60     }
61     p.print(reflect.ValueOf(x))
62     p.printf("\n")
63
64     return
65 }
66
67 // Print prints x to standard output, skipping nil fields.
68 // Print(fset, x) is the same as Fprint(os.Stdout, fset, x,
69 func Print(fset *token.FileSet, x interface{}) error {
70     return Fprint(os.Stdout, fset, x, NotNilFilter)
71 }
72
73 type printer struct {
74     output io.Writer
75     fset   *token.FileSet
76     filter FieldFilter
77     ptrmap map[interface{}]int // *T -> line number
78     indent int           // current indentation le
79     last   byte           // the last byte processe
80     line   int           // current line number
81 }
82
83 var indent = []byte(". ")
84
85 func (p *printer) Write(data []byte) (n int, err error) {
86     var m int
87     for i, b := range data {
88         // invariant: data[0:n] has been written
89         if b == '\n' {
90             m, err = p.output.Write(data[n : i+1])
91             n += m
92             if err != nil {
93                 return
94             }

```

```

95         p.line++
96     } else if p.last == '\n' {
97         _, err = fmt.Fprintf(p.output, "%6d
98         if err != nil {
99             return
100        }
101        for j := p.indent; j > 0; j-- {
102            _, err = p.output.Write(inde
103            if err != nil {
104                return
105            }
106        }
107    }
108    p.last = b
109 }
110 m, err = p.output.Write(data[n:])
111 n += m
112 return
113 }
114
115 // localError wraps locally caught errors so we can distingu
116 // them from genuine panics which we don't want to return as
117 type localError struct {
118     err error
119 }
120
121 // printf is a convenience wrapper that takes care of print
122 func (p *printer) printf(format string, args ...interface{})
123     if _, err := fmt.Fprintf(p, format, args...); err !=
124         panic(localError{err})
125 }
126 }
127
128 // Implementation note: Print is written for AST nodes but c
129 // used to print arbitrary data structures; such a version s
130 // probably be in a different package.
131 //
132 // Note: This code detects (some) cycles created via pointer
133 // not cycles that are created via slices or maps containing
134 // same slice or map. Code for general data structures proba
135 // should catch those as well.
136
137 func (p *printer) print(x reflect.Value) {
138     if !NotNilFilter("", x) {
139         p.printf("nil")
140         return
141     }
142
143     switch x.Kind() {

```

```

144     case reflect.Interface:
145         p.print(x.Elem())
146
147     case reflect.Map:
148         p.printf("%s (len = %d) {\n", x.Type(), x.Len())
149         p.indent++
150         for _, key := range x.MapKeys() {
151             p.print(key)
152             p.printf(": ")
153             p.print(x.MapIndex(key))
154             p.printf("\n")
155         }
156         p.indent--
157         p.printf("}")
158
159     case reflect.Ptr:
160         p.printf("*")
161         // type-checked ASTs may contain cycles - us
162         // to keep track of objects that have been p
163         // already and print the respective line num
164         ptr := x.Interface()
165         if line, exists := p.ptrmap[ptr]; exists {
166             p.printf("(obj @ %d)", line)
167         } else {
168             p.ptrmap[ptr] = p.line
169             p.print(x.Elem())
170         }
171
172     case reflect.Slice:
173         if s, ok := x.Interface().([]byte); ok {
174             p.printf("%#q", s)
175             return
176         }
177         p.printf("%s (len = %d) {\n", x.Type(), x.Len())
178         p.indent++
179         for i, n := 0, x.Len(); i < n; i++ {
180             p.printf("%d: ", i)
181             p.print(x.Index(i))
182             p.printf("\n")
183         }
184         p.indent--
185         p.printf("}")
186
187     case reflect.Struct:
188         p.printf("%s {\n", x.Type())
189         p.indent++
190         t := x.Type()
191         for i, n := 0, t.NumField(); i < n; i++ {
192             name := t.Field(i).Name

```

```

193             value := x.Field(i)
194             if p.filter == nil || p.filter(name,
195                 p.printf("%s: ", name)
196                 p.print(value)
197                 p.printf("\n")
198             }
199         }
200     p.indent--
201     p.printf("}")
202
203     default:
204         v := x.Interface()
205         switch v := v.(type) {
206         case string:
207             // print strings in quotes
208             p.printf("%q", v)
209             return
210         case token.Pos:
211             // position values can be printed ni
212             if p.fset != nil {
213                 p.printf("%s", p.fset.Position(
214                     v))
215             }
216         }
217         // default
218         p.printf("%v", v)
219     }
220 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/ast/resolve.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file implements NewPackage.
6
7 package ast
8
9 import (
10     "fmt"
11     "go/scanner"
12     "go/token"
13     "strconv"
14 )
15
16 type pkgBuilder struct {
17     fset    *token.FileSet
18     errors scanner.ErrorList
19 }
20
21 func (p *pkgBuilder) error(pos token.Pos, msg string) {
22     p.errors.Add(p.fset.Position(pos), msg)
23 }
24
25 func (p *pkgBuilder) errorf(pos token.Pos, format string, ar
26     p.error(pos, fmt.Sprintf(format, args...))
27 }
28
29 func (p *pkgBuilder) declare(scope, altScope *Scope, obj *Ob
30     alt := scope.Insert(obj)
31     if alt == nil && altScope != nil {
32         // see if there is a conflicting declaration
33         alt = altScope.Lookup(obj.Name)
34     }
35     if alt != nil {
36         prevDecl := ""
37         if pos := alt.Pos(); pos.IsValid() {
38             prevDecl = fmt.Sprintf("\n\tprevious
39         }
40         p.error(obj.Pos(), fmt.Sprintf("%s redeclare
41     }
42 }
43
44 func resolve(scope *Scope, ident *Ident) bool {
```

```

45         for ; scope != nil; scope = scope.Outer {
46             if obj := scope.Lookup(ident.Name); obj != n
47                 ident.Obj = obj
48                 return true
49         }
50     }
51     return false
52 }
53
54 // An Importer resolves import paths to package Objects.
55 // The imports map records the packages already imported,
56 // indexed by package id (canonical import path).
57 // An Importer must determine the canonical import path and
58 // check the map to see if it is already present in the impo
59 // If so, the Importer can return the map entry. Otherwise,
60 // Importer should load the package data for the given path
61 // a new *Object (pkg), record pkg in the imports map, and t
62 // return pkg.
63 type Importer func(imports map[string]*Object, path string)
64
65 // NewPackage creates a new Package node from a set of File
66 // unresolved identifiers across files and updates each file
67 // accordingly. If a non-nil importer and universe scope are
68 // used to resolve identifiers not declared in any of the pa
69 // remaining unresolved identifiers are reported as undeclar
70 // belong to different packages, one package name is selecte
71 // different package names are reported and then ignored.
72 // The result is a package node and a scanner.ErrorList if t
73 //
74 func NewPackage(fset *token.FileSet, files map[string]*File,
75     var p pkgBuilder
76     p.fset = fset
77
78     // complete package scope
79     pkgName := ""
80     pkgScope := NewScope(universe)
81     for _, file := range files {
82         // package names must match
83         switch name := file.Name.Name; {
84             case pkgName == "":
85                 pkgName = name
86             case name != pkgName:
87                 p.errorf(file.Package, "package %s;
88                 continue // ignore this file
89         }
90
91         // collect top-level file objects in package
92         for _, obj := range file.Scope.Objects {
93             p.declare(pkgScope, nil, obj)
94         }

```

```

95     }
96
97     // package global mapping of imported package ids to
98     imports := make(map[string]*Object)
99
100    // complete file scopes with imports and resolve ide
101    for _, file := range files {
102        // ignore file if it belongs to a different
103        // (error has already been reported)
104        if file.Name.Name != pkgName {
105            continue
106        }
107
108        // build file scope by processing all import
109        importErrors := false
110        fileScope := NewScope(pkgScope)
111        for _, spec := range file.Imports {
112            if importer == nil {
113                importErrors = true
114                continue
115            }
116            path, _ := strconv.Unquote(spec.Path)
117            pkg, err := importer(imports, path)
118            if err != nil {
119                p.errorf(spec.Path.Pos(), "c
120                importErrors = true
121                continue
122            }
123            // TODO(gri) If a local package name
124            // global identifier resolution coul
125            // import failed. Consider adjusting
126
127            // local name overrides imported pac
128            name := pkg.Name
129            if spec.Name != nil {
130                name = spec.Name.Name
131            }
132
133            // add import to file scope
134            if name == "." {
135                // merge imported scope with
136                for _, obj := range pkg.Data
137                    p.declare(fileScope,
138                }
139            } else {
140                // declare imported package
141                // (do not re-use pkg in the
142                // a new object instead; the
143                // for different files)

```

```

144         obj := NewObj(Pkg, name)
145         obj.Decl = spec
146         obj.Data = pkg.Data
147         p.declare(fileScope, pkgScop
148     }
149 }
150
151 // resolve identifiers
152 if importErrors {
153     // don't use the universe scope with
154     // (objects in the universe may be s
155     // with missing imports, identifiers
156     // incorrectly to universe objects)
157     pkgScope.Outer = nil
158 }
159 i := 0
160 for _, ident := range file.Unresolved {
161     if !resolve(fileScope, ident) {
162         p.errorf(ident.Pos(), "undec
163             file.Unresolved[i] = ident
164             i++
165     }
166 }
167 }
168 file.Unresolved = file.Unresolved[0:i]
169 pkgScope.Outer = universe // reset universe
170 }
171
172 p.errors.Sort()
173 return &Package{pkgName, pkgScope, imports, files},
174 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/ast/scope.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file implements scopes and the objects they contain.
6
7 package ast
8
9 import (
10     "bytes"
11     "fmt"
12     "go/token"
13 )
14
15 // A Scope maintains the set of named language entities declared
16 // in the scope and a link to the immediately surrounding (outer)
17 // scope.
18 //
19 type Scope struct {
20     Outer *Scope
21     Objects map[string]*Object
22 }
23
24 // NewScope creates a new scope nested in the outer scope.
25 func NewScope(outer *Scope) *Scope {
26     const n = 4 // initial scope capacity
27     return &Scope{outer, make(map[string]*Object, n)}
28 }
29
30 // Lookup returns the object with the given name if it is
31 // found in scope s, otherwise it returns nil. Outer scopes
32 // are ignored.
33 //
34 func (s *Scope) Lookup(name string) *Object {
35     return s.Objects[name]
36 }
37
38 // Insert attempts to insert a named object obj into the scope s.
39 // If the scope already contains an object alt with the same name,
40 // Insert leaves the scope unchanged and returns alt. Otherwise,
41 // it inserts obj and returns nil.
42 //
43 func (s *Scope) Insert(obj *Object) (alt *Object) {
44     if alt = s.Objects[obj.Name]; alt == nil {
```

```

45             s.Objects[obj.Name] = obj
46         }
47         return
48     }
49
50 // Debugging support
51 func (s *Scope) String() string {
52     var buf bytes.Buffer
53     fmt.Fprintf(&buf, "scope %p {"", s)
54     if s != nil && len(s.Objects) > 0 {
55         fmt.Fprintln(&buf)
56         for _, obj := range s.Objects {
57             fmt.Fprintf(&buf, "\t%s %s\n", obj.K
58         }
59     }
60     fmt.Fprintf(&buf, "}\n")
61     return buf.String()
62 }
63
64 // -----
65 // Objects
66
67 // TODO(gri) Consider replacing the Object struct with an in
68 //             and a corresponding set of object implementatio
69
70 // An Object describes a named language entity such as a pac
71 // constant, type, variable, function (incl. methods), or la
72 //
73 // The Data fields contains object-specific data:
74 //
75 //     Kind      Data type      Data value
76 //     Pkg        *Scope         package scope
77 //     Con        int             iota for the respective declara
78 //     Con        != nil          constant value
79 //
80 type Object struct {
81     Kind ObjKind
82     Name string           // declared name
83     Decl interface{}     // corresponding Field, XxxSpec, Fu
84     Data interface{}     // object-specific data; or nil
85     Type interface{}     // place holder for type informatio
86 }
87
88 // NewObj creates a new object of a given kind and name.
89 func NewObj(kind ObjKind, name string) *Object {
90     return &Object{Kind: kind, Name: name}
91 }
92
93 // Pos computes the source position of the declaration of an
94 // The result may be an invalid position if it cannot be com

```

```

95 // (obj.Decl may be nil or not correct).
96 func (obj *Object) Pos() token.Pos {
97     name := obj.Name
98     switch d := obj.Decl.(type) {
99     case *Field:
100         for _, n := range d.Names {
101             if n.Name == name {
102                 return n.Pos()
103             }
104         }
105     case *ImportSpec:
106         if d.Name != nil && d.Name.Name == name {
107             return d.Name.Pos()
108         }
109         return d.Path.Pos()
110     case *ValueSpec:
111         for _, n := range d.Names {
112             if n.Name == name {
113                 return n.Pos()
114             }
115         }
116     case *TypeSpec:
117         if d.Name.Name == name {
118             return d.Name.Pos()
119         }
120     case *FuncDecl:
121         if d.Name.Name == name {
122             return d.Name.Pos()
123         }
124     case *LabeledStmt:
125         if d.Label.Name == name {
126             return d.Label.Pos()
127         }
128     case *AssignStmt:
129         for _, x := range d.Lhs {
130             if ident, isIdent := x.(*Ident); isI
131                 return ident.Pos()
132         }
133     }
134     case *Scope:
135         // predeclared object - nothing to do for no
136     }
137     return token.NoPos
138 }
139
140 // ObKind describes what an object represents.
141 type ObKind int
142
143 // The list of possible Object kinds.

```

```
144 const (
145     Bad ObjKind = iota // for error handling
146     Pkg          // package
147     Con          // constant
148     Typ          // type
149     Var          // variable
150     Fun          // function or method
151     Lbl          // label
152 )
153
154 var objKindStrings = [...]string{
155     Bad: "bad",
156     Pkg: "package",
157     Con: "const",
158     Typ: "type",
159     Var: "var",
160     Fun: "func",
161     Lbl: "label",
162 }
163
164 func (kind ObjKind) String() string { return objKindStrings[
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/ast/walk.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package ast
6
7 import "fmt"
8
9 // A Visitor's Visit method is invoked for each node encount
10 // If the result visitor w is not nil, Walk visits each of t
11 // of node with the visitor w, followed by a call of w.Visit
12 type Visitor interface {
13     Visit(node Node) (w Visitor)
14 }
15
16 // Helper functions for common node lists. They may be empty
17
18 func walkIdentList(v Visitor, list []*Ident) {
19     for _, x := range list {
20         Walk(v, x)
21     }
22 }
23
24 func walkExprList(v Visitor, list []Expr) {
25     for _, x := range list {
26         Walk(v, x)
27     }
28 }
29
30 func walkStmtList(v Visitor, list []Stmt) {
31     for _, x := range list {
32         Walk(v, x)
33     }
34 }
35
36 func walkDeclList(v Visitor, list []Decl) {
37     for _, x := range list {
38         Walk(v, x)
39     }
40 }
41
42 // TODO(gri): Investigate if providing a closure to Walk lea
43 // simpler use (and may help eliminate Inspect in
44
```

```

45 // Walk traverses an AST in depth-first order: It starts by
46 // v.Visit(node); node must not be nil. If the visitor w ret
47 // v.Visit(node) is not nil, Walk is invoked recursively wit
48 // w for each of the non-nil children of node, followed by a
49 // w.Visit(nil).
50 //
51 func Walk(v Visitor, node Node) {
52     if v = v.Visit(node); v == nil {
53         return
54     }
55
56     // walk children
57     // (the order of the cases matches the order
58     // of the corresponding node types in ast.go)
59     switch n := node.(type) {
60     // Comments and fields
61     case *Comment:
62         // nothing to do
63
64     case *CommentGroup:
65         for _, c := range n.List {
66             Walk(v, c)
67         }
68
69     case *Field:
70         if n.Doc != nil {
71             Walk(v, n.Doc)
72         }
73         walkIdentList(v, n.Names)
74         Walk(v, n.Type)
75         if n.Tag != nil {
76             Walk(v, n.Tag)
77         }
78         if n.Comment != nil {
79             Walk(v, n.Comment)
80         }
81
82     case *FieldList:
83         for _, f := range n.List {
84             Walk(v, f)
85         }
86
87     // Expressions
88     case *BadExpr, *Ident, *BasicLit:
89         // nothing to do
90
91     case *Ellipsis:
92         if n.Elt != nil {
93             Walk(v, n.Elt)
94         }

```

```
95
96     case *FuncLit:
97         Walk(v, n.Type)
98         Walk(v, n.Body)
99
100    case *CompositeLit:
101        if n.Type != nil {
102            Walk(v, n.Type)
103        }
104        walkExprList(v, n.Elts)
105
106    case *ParenExpr:
107        Walk(v, n.X)
108
109    case *SelectorExpr:
110        Walk(v, n.X)
111        Walk(v, n.Sel)
112
113    case *IndexExpr:
114        Walk(v, n.X)
115        Walk(v, n.Index)
116
117    case *SliceExpr:
118        Walk(v, n.X)
119        if n.Low != nil {
120            Walk(v, n.Low)
121        }
122        if n.High != nil {
123            Walk(v, n.High)
124        }
125
126    case *TypeAssertExpr:
127        Walk(v, n.X)
128        if n.Type != nil {
129            Walk(v, n.Type)
130        }
131
132    case *CallExpr:
133        Walk(v, n.Fun)
134        walkExprList(v, n.Args)
135
136    case *StarExpr:
137        Walk(v, n.X)
138
139    case *UnaryExpr:
140        Walk(v, n.X)
141
142    case *BinaryExpr:
143        Walk(v, n.X)
```

```
144         Walk(v, n.Y)
145
146     case *KeyValueExpr:
147         Walk(v, n.Key)
148         Walk(v, n.Value)
149
150     // Types
151     case *ArrayType:
152         if n.Len != nil {
153             Walk(v, n.Len)
154         }
155         Walk(v, n.Elt)
156
157     case *StructType:
158         Walk(v, n.Fields)
159
160     case *FuncType:
161         Walk(v, n.Params)
162         if n.Results != nil {
163             Walk(v, n.Results)
164         }
165
166     case *InterfaceType:
167         Walk(v, n.Methods)
168
169     case *MapType:
170         Walk(v, n.Key)
171         Walk(v, n.Value)
172
173     case *ChanType:
174         Walk(v, n.Value)
175
176     // Statements
177     case *BadStmt:
178         // nothing to do
179
180     case *DeclStmt:
181         Walk(v, n.Decl)
182
183     case *EmptyStmt:
184         // nothing to do
185
186     case *LabeledStmt:
187         Walk(v, n.Label)
188         Walk(v, n.Stmt)
189
190     case *ExprStmt:
191         Walk(v, n.X)
192
```

```
193     case *SendStmt:
194         Walk(v, n.Chan)
195         Walk(v, n.Value)
196
197     case *IncDecStmt:
198         Walk(v, n.X)
199
200     case *AssignStmt:
201         walkExprList(v, n.Lhs)
202         walkExprList(v, n.Rhs)
203
204     case *GoStmt:
205         Walk(v, n.Call)
206
207     case *DeferStmt:
208         Walk(v, n.Call)
209
210     case *ReturnStmt:
211         walkExprList(v, n.Results)
212
213     case *BranchStmt:
214         if n.Label != nil {
215             Walk(v, n.Label)
216         }
217
218     case *BlockStmt:
219         walkStmtList(v, n.List)
220
221     case *IfStmt:
222         if n.Init != nil {
223             Walk(v, n.Init)
224         }
225         Walk(v, n.Cond)
226         Walk(v, n.Body)
227         if n.Else != nil {
228             Walk(v, n.Else)
229         }
230
231     case *CaseClause:
232         walkExprList(v, n.List)
233         walkStmtList(v, n.Body)
234
235     case *SwitchStmt:
236         if n.Init != nil {
237             Walk(v, n.Init)
238         }
239         if n.Tag != nil {
240             Walk(v, n.Tag)
241         }
242         Walk(v, n.Body)
```

```

243
244     case *TypeSwitchStmt:
245         if n.Init != nil {
246             Walk(v, n.Init)
247         }
248         Walk(v, n.Assign)
249         Walk(v, n.Body)
250
251     case *CommClause:
252         if n.Comm != nil {
253             Walk(v, n.Comm)
254         }
255         walkStmtList(v, n.Body)
256
257     case *SelectStmt:
258         Walk(v, n.Body)
259
260     case *ForStmt:
261         if n.Init != nil {
262             Walk(v, n.Init)
263         }
264         if n.Cond != nil {
265             Walk(v, n.Cond)
266         }
267         if n.Post != nil {
268             Walk(v, n.Post)
269         }
270         Walk(v, n.Body)
271
272     case *RangeStmt:
273         Walk(v, n.Key)
274         if n.Value != nil {
275             Walk(v, n.Value)
276         }
277         Walk(v, n.X)
278         Walk(v, n.Body)
279
280     // Declarations
281     case *ImportSpec:
282         if n.Doc != nil {
283             Walk(v, n.Doc)
284         }
285         if n.Name != nil {
286             Walk(v, n.Name)
287         }
288         Walk(v, n.Path)
289         if n.Comment != nil {
290             Walk(v, n.Comment)
291         }

```

```

292
293     case *ValueSpec:
294         if n.Doc != nil {
295             Walk(v, n.Doc)
296         }
297         walkIdentList(v, n.Names)
298         if n.Type != nil {
299             Walk(v, n.Type)
300         }
301         walkExprList(v, n.Values)
302         if n.Comment != nil {
303             Walk(v, n.Comment)
304         }
305
306     case *TypeSpec:
307         if n.Doc != nil {
308             Walk(v, n.Doc)
309         }
310         Walk(v, n.Name)
311         Walk(v, n.Type)
312         if n.Comment != nil {
313             Walk(v, n.Comment)
314         }
315
316     case *BadDecl:
317         // nothing to do
318
319     case *GenDecl:
320         if n.Doc != nil {
321             Walk(v, n.Doc)
322         }
323         for _, s := range n.Specs {
324             Walk(v, s)
325         }
326
327     case *FuncDecl:
328         if n.Doc != nil {
329             Walk(v, n.Doc)
330         }
331         if n.Recv != nil {
332             Walk(v, n.Recv)
333         }
334         Walk(v, n.Name)
335         Walk(v, n.Type)
336         if n.Body != nil {
337             Walk(v, n.Body)
338         }
339
340 // Files and packages

```

```

341     case *File:
342         if n.Doc != nil {
343             Walk(v, n.Doc)
344         }
345         Walk(v, n.Name)
346         walkDeclList(v, n.Decls)
347         for _, g := range n.Comments {
348             Walk(v, g)
349         }
350         // don't walk n.Comments - they have been
351         // visited already through the individual
352         // nodes
353
354     case *Package:
355         for _, f := range n.Files {
356             Walk(v, f)
357         }
358
359     default:
360         fmt.Printf("ast.Walk: unexpected node type %
361             panic("ast.Walk")
362     }
363
364     v.Visit(nil)
365 }
366
367 type inspector func(Node) bool
368
369 func (f inspector) Visit(node Node) Visitor {
370     if f(node) {
371         return f
372     }
373     return nil
374 }
375
376 // Inspect traverses an AST in depth-first order: It starts
377 // f(node); node must not be nil. If f returns true, Inspect
378 // for all the non-nil children of node, recursively.
379 //
380 func Inspect(node Node, f func(Node) bool) {
381     Walk(inspector(f), node)
382 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/build/build.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package build
6
7 import (
8     "bytes"
9     "errors"
10    "fmt"
11    "go/ast"
12    "go/doc"
13    "go/parser"
14    "go/token"
15    "io"
16    "io/ioutil"
17    "log"
18    "os"
19    pathpkg "path"
20    "path/filepath"
21    "runtime"
22    "sort"
23    "strconv"
24    "strings"
25    "unicode"
26 )
27
28 // A Context specifies the supporting context for a build.
29 type Context struct {
30     GOARCH    string // target architecture
31     GOOS      string // target operating system
32     GOROOT    string // Go root
33     GOPATH    string // Go path
34     CgoEnabled bool   // whether cgo can be used
35     BuildTags []string // additional tags to recognize
36     UseAllFiles bool   // use files regardless of +bui
37     Compiler  string // compiler to assume when comp
38
39     // By default, Import uses the operating system's fi
40     // to read directories and files. To read from othe
41     // callers can set the following functions. They al
42     // behaviors that use the local file system, so clie
43     // the functions whose behaviors they wish to change
44
```

```

45     // JoinPath joins the sequence of path fragments into a string
46     // If JoinPath is nil, Import uses filepath.Join.
47     JoinPath func(elem ...string) string
48
49     // SplitPathList splits the path list into a slice of strings
50     // If SplitPathList is nil, Import uses filepath.SplitList.
51     SplitPathList func(list string) []string
52
53     // IsAbsPath reports whether path is an absolute path
54     // If IsAbsPath is nil, Import uses filepath.IsAbs.
55     IsAbsPath func(path string) bool
56
57     // IsDir reports whether the path names a directory.
58     // If IsDir is nil, Import calls os.Stat and uses the result.
59     IsDir func(path string) bool
60
61     // HasSubdir reports whether dir is a subdirectory of root.
62     // (perhaps multiple levels below) root.
63     // If so, HasSubdir sets rel to a slash-separated path
64     // that can be joined to root to produce a path equivalent to dir.
65     // If HasSubdir is nil, Import uses an implementation of
66     // filepath.EvalSymlinks.
67     HasSubdir func(root, dir string) (rel string, ok bool)
68
69     // ReadDir returns a slice of os.FileInfo, sorted by name,
70     // describing the content of the named directory.
71     // If ReadDir is nil, Import uses io.ReadDir.
72     ReadDir func(dir string) (fi []os.FileInfo, err error)
73
74     // OpenFile opens a file (not a directory) for reading.
75     // If OpenFile is nil, Import uses os.Open.
76     OpenFile func(path string) (r io.ReadCloser, err error)
77 }
78
79 // joinPath calls ctxt.JoinPath (if not nil) or else filepath.Join.
80 func (ctxt *Context) joinPath(elem ...string) string {
81     if f := ctxt.JoinPath; f != nil {
82         return f(elem...)
83     }
84     return filepath.Join(elem...)
85 }
86
87 // splitPathList calls ctxt.SplitPathList (if not nil) or else
88 // filepath.SplitList.
89 func (ctxt *Context) splitPathList(s string) []string {
90     if f := ctxt.SplitPathList; f != nil {
91         return f(s)
92     }
93     return filepath.SplitList(s)
94 }

```

```

95 // isAbsPath calls ctxt.IsAbsSPath (if not nil) or else file
96 func (ctxt *Context) isAbsPath(path string) bool {
97     if f := ctxt.IsAbsPath; f != nil {
98         return f(path)
99     }
100     return filepath.IsAbs(path)
101 }
102
103 // isDir calls ctxt.IsDir (if not nil) or else uses os.Stat.
104 func (ctxt *Context) isDir(path string) bool {
105     if f := ctxt.IsDir; f != nil {
106         return f(path)
107     }
108     fi, err := os.Stat(path)
109     return err == nil && fi.IsDir()
110 }
111
112 // hasSubdir calls ctxt.HasSubdir (if not nil) or else uses
113 // the local file system to answer the question.
114 func (ctxt *Context) hasSubdir(root, dir string) (rel string
115     if f := ctxt.HasSubdir; f != nil {
116         return f(root, dir)
117     }
118
119     if p, err := filepath.EvalSymlinks(root); err == nil
120         root = p
121     }
122     if p, err := filepath.EvalSymlinks(dir); err == nil
123         dir = p
124     }
125     const sep = string(filepath.Separator)
126     root = filepath.Clean(root)
127     if !strings.HasSuffix(root, sep) {
128         root += sep
129     }
130     dir = filepath.Clean(dir)
131     if !strings.HasPrefix(dir, root) {
132         return "", false
133     }
134     return filepath.ToSlash(dir[len(root):]), true
135 }
136
137 // readDir calls ctxt.ReadDir (if not nil) or else ioutil.Re
138 func (ctxt *Context) readDir(path string) ([]os.FileInfo, er
139     if f := ctxt.ReadDir; f != nil {
140         return f(path)
141     }
142     return ioutil.ReadDir(path)
143 }

```

```

144
145 // openFile calls ctxt.OpenFile (if not nil) or else os.Open
146 func (ctxt *Context) openFile(path string) (io.ReadCloser, e
147     if fn := ctxt.OpenFile; fn != nil {
148         return fn(path)
149     }
150
151     f, err := os.Open(path)
152     if err != nil {
153         return nil, err // nil interface
154     }
155     return f, nil
156 }
157
158 // isFile determines whether path is a file by trying to ope
159 // It reuses openFile instead of adding another function to
160 // list in Context.
161 func (ctxt *Context) isFile(path string) bool {
162     f, err := ctxt.openFile(path)
163     if err != nil {
164         return false
165     }
166     f.Close()
167     return true
168 }
169
170 // gopath returns the list of Go path directories.
171 func (ctxt *Context) gopath() []string {
172     var all []string
173     for _, p := range ctxt.splitPathList(ctxt.GOPATH) {
174         if p == "" || p == ctxt.GOROOT {
175             // Empty paths are uninteresting.
176             // If the path is the GOROOT, ignore
177             // People sometimes set GOPATH=$GORO
178             // but would cause us to find packag
179             // like "pkg/math".
180             // Do not get confused by this commo
181             continue
182         }
183         all = append(all, p)
184     }
185     return all
186 }
187
188 // SrcDirs returns a list of package source root directories
189 // It draws from the current Go root and Go path but omits d
190 // that do not exist.
191 func (ctxt *Context) SrcDirs() []string {
192     var all []string

```

```

193     if ctxt.GOROOT != "" {
194         dir := ctxt.joinPath(ctxt.GOROOT, "src", "pk
195         if ctxt.isDir(dir) {
196             all = append(all, dir)
197         }
198     }
199     for _, p := range ctxt.gopath() {
200         dir := ctxt.joinPath(p, "src")
201         if ctxt.isDir(dir) {
202             all = append(all, dir)
203         }
204     }
205     return all
206 }
207
208 // Default is the default Context for builds.
209 // It uses the GOARCH, GOOS, GOROOT, and GOPATH environment
210 // if set, or else the compiled code's GOARCH, GOOS, and GOR
211 var Default Context = defaultContext()
212
213 var cgoEnabled = map[string]bool{
214     "darwin/386":    true,
215     "darwin/amd64": true,
216     "linux/386":     true,
217     "linux/amd64":  true,
218     "freebsd/386":  true,
219     "freebsd/amd64": true,
220     "windows/386":  true,
221     "windows/amd64": true,
222 }
223
224 func defaultContext() Context {
225     var c Context
226
227     c.GOARCH = envOr("GOARCH", runtime.GOARCH)
228     c.GOOS = envOr("GOOS", runtime.GOOS)
229     c.GOROOT = runtime.GOROOT()
230     c.GOPATH = envOr("GOPATH", "")
231     c.Compiler = runtime.Compiler
232
233     switch os.Getenv("CGO_ENABLED") {
234     case "1":
235         c.CgoEnabled = true
236     case "0":
237         c.CgoEnabled = false
238     default:
239         c.CgoEnabled = cgoEnabled[c.GOOS+"/"+c.GOARCH]
240     }
241
242     return c

```

```

243 }
244
245 func envOr(name, def string) string {
246     s := os.Getenv(name)
247     if s == "" {
248         return def
249     }
250     return s
251 }
252
253 // An ImportMode controls the behavior of the Import method.
254 type ImportMode uint
255
256 const (
257     // If FindOnly is set, Import stops after locating t
258     // that should contain the sources for a package. I
259     // read any files in the directory.
260     FindOnly ImportMode = 1 << iota
261
262     // If AllowBinary is set, Import can be satisfied by
263     // package object without corresponding sources.
264     AllowBinary
265 )
266
267 // A Package describes the Go package found in a directory.
268 type Package struct {
269     Dir          string // directory containing package so
270     Name         string // package name
271     Doc          string // documentation synopsis
272     ImportPath   string // import path of package (" " if u
273     Root         string // root of Go tree where this pack
274     SrcRoot      string // package source root directory (
275     PkgRoot      string // package install root directory
276     BinDir       string // command install directory (" " i
277     Goroot       bool   // package found in Go root
278     PkgObj       string // installed .a file
279
280     // Source files
281     GoFiles      []string // .go source files (excluding Cg
282     CgoFiles     []string // .go source files that import "
283     CFiles       []string // .c source files
284     HFiles       []string // .h source files
285     SFiles       []string // .s source files
286     SysoFiles    []string // .syso system object files to a
287
288     // Cgo directives
289     CgoPkgConfig []string // Cgo pkg-config directives
290     CgoCFLAGS    []string // Cgo CFLAGS directives
291     CgoLDFLAGS   []string // Cgo LDFLAGS directives

```

```

292
293 // Dependency information
294 Imports []string // imports fro
295 ImportPos map[string][]token.Position // line inform
296
297 // Test information
298 TestGoFiles []string // _test.
299 TestImports []string // import
300 TestImportPos map[string][]token.Position // line i
301 XTestGoFiles []string // _test.
302 XTestImports []string // import
303 XTestImportPos map[string][]token.Position // line i
304 }
305
306 // IsCommand reports whether the package is considered a
307 // command to be installed (not just a library).
308 // Packages named "main" are treated as commands.
309 func (p *Package) IsCommand() bool {
310     return p.Name == "main"
311 }
312
313 // ImportDir is like Import but processes the Go package fou
314 // the named directory.
315 func (ctxt *Context) ImportDir(dir string, mode ImportMode)
316     return ctxt.Import(".", dir, mode)
317 }
318
319 // NoGoError is the error used by Import to describe a direc
320 // containing no Go source files.
321 type NoGoError struct {
322     Dir string
323 }
324
325 func (e *NoGoError) Error() string {
326     return "no Go source files in " + e.Dir
327 }
328
329 // Import returns details about the Go package named by the
330 // interpreting local import paths relative to the srcDir di
331 // If the path is a local import path naming a package that
332 // using a standard import path, the returned package will s
333 // to that path.
334 //
335 // In the directory containing the package, .go, .c, .h, and
336 // considered part of the package except for:
337 //
338 // - .go files in package documentation
339 // - files starting with _ or . (likely editor temporar
340 // - files with build constraints not satisfied by the

```

```

341 //
342 // If an error occurs, Import returns a non-nil error also r
343 // *Package containing partial information.
344 //
345 func (ctxt *Context) Import(path string, srcDir string, mode
346     p := &Package{
347         ImportPath: path,
348     }
349
350     var pkga string
351     var pkgerr error
352     switch ctxt.Compiler {
353     case "gccgo":
354         dir, elem := pathpkg.Split(p.ImportPath)
355         pkga = "pkg/gccgo/" + dir + "lib" + elem + "
356     case "gc":
357         pkga = "pkg/" + ctxt.GOOS + "_" + ctxt.GOARC
358     default:
359         // Save error for end of function.
360         pkgerr = fmt.Errorf("import %q: unknown comp
361     }
362
363     binaryOnly := false
364     if IsLocalImport(path) {
365         pkga = "" // local imports have no installed
366         if srcDir == "" {
367             return p, fmt.Errorf("import %q: imp
368         }
369         if !ctxt.isAbsPath(path) {
370             p.Dir = ctxt.joinPath(srcDir, path)
371         }
372         // Determine canonical import path, if any.
373         if ctxt.GOROOT != "" {
374             root := ctxt.joinPath(ctxt.GOROOT, "
375             if sub, ok := ctxt.hasSubdir(root, p
376                 p.Goroot = true
377                 p.ImportPath = sub
378                 p.Root = ctxt.GOROOT
379                 goto Found
380             }
381         }
382         all := ctxt.gopath()
383         for i, root := range all {
384             rootsrc := ctxt.joinPath(root, "src"
385             if sub, ok := ctxt.hasSubdir(rootsrc
386                 // We found a potential impo
387                 // but check that using it w
388                 // else first.
389                 if ctxt.GOROOT != "" {
390                     if dir := ctxt.joinP

```

```

391                                     goto Found
392                                     }
393                                 }
394                                 for _, earlyRoot := range al
395                                     if dir := ctxt.joinP
396                                         goto Found
397                                     }
398                                 }
399
400                                 // sub would not name some o
401                                 // Record it.
402                                 p.ImportPath = sub
403                                 p.Root = root
404                                 goto Found
405                             }
406                         }
407                         // It's okay that we didn't find a root cont
408                         // Keep going with the information we have.
409                     } else {
410                         if strings.HasPrefix(path, "/") {
411                             return p, fmt.Errorf("import %q: can
412                         }
413                         // Determine directory from import path.
414                         if ctxt.GOROOT != "" {
415                             dir := ctxt.joinPath(ctxt.GOROOT, "s
416                             isDir := ctxt.isDir(dir)
417                             binaryOnly = !isDir && mode&AllowBin
418                             if isDir || binaryOnly {
419                                 p.Dir = dir
420                                 p.Goroot = true
421                                 p.Root = ctxt.GOROOT
422                                 goto Found
423                             }
424                         }
425                         for _, root := range ctxt.gopath() {
426                             dir := ctxt.joinPath(root, "src", pa
427                             isDir := ctxt.isDir(dir)
428                             binaryOnly = !isDir && mode&AllowBin
429                             if isDir || binaryOnly {
430                                 p.Dir = dir
431                                 p.Root = root
432                                 goto Found
433                             }
434                         }
435                         return p, fmt.Errorf("import %q: cannot find
436                     }
437
438 Found:
439     if p.Root != "" {

```

```

440         if p.Goroot {
441             p.SrcRoot = ctxt.joinPath(p.Root, "s
442         } else {
443             p.SrcRoot = ctxt.joinPath(p.Root, "s
444         }
445         p.PkgRoot = ctxt.joinPath(p.Root, "pkg")
446         p.BinDir = ctxt.joinPath(p.Root, "bin")
447         if pkga != "" {
448             p.PkgObj = ctxt.joinPath(p.Root, pkg
449         }
450     }
451
452     if mode&FindOnly != 0 {
453         return p, pkgerr
454     }
455     if binaryOnly && (mode&AllowBinary) != 0 {
456         return p, pkgerr
457     }
458
459     dirs, err := ctxt.readDir(p.Dir)
460     if err != nil {
461         return p, err
462     }
463
464     var Sfiles []string // files with ".S" (capital S)
465     var firstFile string
466     imported := make(map[string][]token.Position)
467     testImported := make(map[string][]token.Position)
468     xTestImported := make(map[string][]token.Position)
469     fset := token.NewFileSet()
470     for _, d := range dirs {
471         if d.IsDir() {
472             continue
473         }
474         name := d.Name()
475         if strings.HasPrefix(name, "_") ||
476             strings.HasPrefix(name, ".") {
477             continue
478         }
479         if !ctxt.UseAllFiles && !ctxt.goodOSArchFile
480             continue
481     }
482
483     i := strings.LastIndex(name, ".")
484     if i < 0 {
485         i = len(name)
486     }
487     ext := name[i:]
488     switch ext {

```

```

489     case ".go", ".c", ".s", ".h", ".S":
490         // tentatively okay - read to make s
491     case ".syso":
492         // binary objects to add to package
493         // Likely of the form foo_windows.sy
494         // the name was vetted above with go
495         p.SysoFiles = append(p.SysoFiles, na
496         continue
497     default:
498         // skip
499         continue
500     }
501
502     filename := ctxt.joinPath(p.Dir, name)
503     f, err := ctxt.openFile(filename)
504     if err != nil {
505         return p, err
506     }
507     data, err := ioutil.ReadAll(f)
508     f.Close()
509     if err != nil {
510         return p, fmt.Errorf("read %s: %v",
511     }
512
513     // Look for +build comments to accept or rej
514     if !ctxt.UseAllFiles && !ctxt.shouldBuild(da
515         continue
516     }
517
518     // Going to save the file.  For non-Go files
519     switch ext {
520     case ".c":
521         p.CFiles = append(p.CFiles, name)
522         continue
523     case ".h":
524         p.HFiles = append(p.HFiles, name)
525         continue
526     case ".s":
527         p.SFiles = append(p.SFiles, name)
528         continue
529     case ".S":
530         Sfiles = append(Sfiles, name)
531         continue
532     }
533
534     pf, err := parser.ParseFile(fset, filename,
535     if err != nil {
536         return p, err
537     }
538

```

```

539     pkg := string(pf.Name.Name)
540     if pkg == "documentation" {
541         continue
542     }
543
544     isTest := strings.HasSuffix(name, "_test.go")
545     isXTest := false
546     if isTest && strings.HasSuffix(pkg, "_test")
547         isXTest = true
548         pkg = pkg[:len(pkg)-len("_test")]
549     }
550
551     if p.Name == "" {
552         p.Name = pkg
553         firstFile = name
554     } else if pkg != p.Name {
555         return p, fmt.Errorf("found packages
556     }
557     if pf.Doc != nil && p.Doc == "" {
558         p.Doc = doc.Synopsis(pf.Doc.Text())
559     }
560
561     // Record imports and information about cgo.
562     isCgo := false
563     for _, decl := range pf.Decls {
564         d, ok := decl.(*ast.GenDecl)
565         if !ok {
566             continue
567         }
568         for _, dspec := range d.Specs {
569             spec, ok := dspec.(*ast.Impo
570             if !ok {
571                 continue
572             }
573             quoted := string(spec.Path.V
574             path, err := strconv.Unquote
575             if err != nil {
576                 log.Panicf("%s: pars
577             }
578             if isXTest {
579                 xTestImported[path]
580             } else if isTest {
581                 testImported[path] =
582             } else {
583                 imported[path] = app
584             }
585             if path == "C" {
586                 if isTest {
587                     return p, fr

```

```

588                                     }
589                                     cg := spec.Doc
590                                     if cg == nil && len(
591                                         cg = d.Doc
592                                     }
593                                     if cg != nil {
594                                         if err := ct
595                                             retu
596                                         }
597                                     }
598                                     isCgo = true
599                                 }
600                             }
601                         }
602                         if isCgo {
603                             if ctxt.CgoEnabled {
604                                 p.CgoFiles = append(p.CgoFil
605                             }
606                         } else if isXTest {
607                             p.XTestGoFiles = append(p.XTestGoFil
608                         } else if isTest {
609                             p.TestGoFiles = append(p.TestGoFiles
610                         } else {
611                             p.GoFiles = append(p.GoFiles, name)
612                         }
613                     }
614                     if p.Name == "" {
615                         return p, &NoGoError{p.Dir}
616                     }
617
618                     p.Imports, p.ImportPos = cleanImports(imported)
619                     p.TestImports, p.TestImportPos = cleanImports(testIr
620                     p.XTestImports, p.XTestImportPos = cleanImports(xTes
621
622                     // add the .S files only if we are using cgo
623                     // (which means gcc will compile them).
624                     // The standard assemblers expect .s files.
625                     if len(p.CgoFiles) > 0 {
626                         p.SFiles = append(p.SFiles, Sfiles...)
627                         sort.Strings(p.SFiles)
628                     }
629
630                     return p, pkgerr
631                 }
632
633 func cleanImports(m map[string][]token.Position) ([]string,
634     all := make([]string, 0, len(m))
635     for path := range m {
636         all = append(all, path)

```

```

637         }
638         sort.Strings(all)
639         return all, m
640     }
641
642     // Import is shorthand for Default.Import.
643     func Import(path, srcDir string, mode ImportMode) (*Package,
644         return Default.Import(path, srcDir, mode)
645     }
646
647     // ImportDir is shorthand for Default.ImportDir.
648     func ImportDir(dir string, mode ImportMode) (*Package, error
649         return Default.ImportDir(dir, mode)
650     }
651
652     var slashslash = []byte("//")
653
654     // shouldBuild reports whether it is okay to use this file,
655     // The rule is that in the file's leading run of // comments
656     // and blank lines, which must be followed by a blank line
657     // (to avoid including a Go package clause doc comment),
658     // lines beginning with '// +build' are taken as build direc
659     //
660     // The file is accepted only if each such line lists somethi
661     // matching the file.  For example:
662     //
663     //     // +build windows linux
664     //
665     // marks the file as applicable only on Windows and Linux.
666     //
667     func (ctxt *Context) shouldBuild(content []byte) bool {
668         // Pass 1. Identify leading run of // comments and b
669         // which must be followed by a blank line.
670         end := 0
671         p := content
672         for len(p) > 0 {
673             line := p
674             if i := bytes.IndexByte(line, '\n'); i >= 0
675                 line, p = line[:i], p[i+1:]
676             } else {
677                 p = p[len(p):]
678             }
679             line = bytes.TrimSpace(line)
680             if len(line) == 0 { // Blank line
681                 end = cap(content) - cap(line) // &l
682                 continue
683             }
684             if !bytes.HasPrefix(line, slashslash) { // N
685                 break
686             }

```

```

687     }
688     content = content[:end]
689
690     // Pass 2. Process each line in the run.
691     p = content
692     for len(p) > 0 {
693         line := p
694         if i := bytes.IndexByte(line, '\n'); i >= 0
695             line, p = line[:i], p[i+1:]
696         } else {
697             p = p[len(p):]
698         }
699         line = bytes.TrimSpace(line)
700         if bytes.HasPrefix(line, slashslash) {
701             line = bytes.TrimSpace(line[len(slas
702             if len(line) > 0 && line[0] == '+' {
703                 // Looks like a comment +lin
704                 f := strings.Fields(string(l
705                 if f[0] == "+build" {
706                     ok := false
707                     for _, tok := range
708                         if ctxt.matc
709                             ok =
710                             brea
711                     }
712                 }
713                 if !ok {
714                     return false
715                 }
716             }
717         }
718     }
719 }
720 return true // everything matches
721 }
722
723 // saveCgo saves the information from the #cgo lines in the
724 // These lines set CFLAGS and LDFLAGS and pkg-config directi
725 // the way cgo's C code is built.
726 //
727 // TODO(rsc): This duplicates code in cgo.
728 // Once the dust settles, remove this code from cgo.
729 func (ctxt *Context) saveCgo(filename string, di *Package, c
730     text := cg.Text()
731     for _, line := range strings.Split(text, "\n") {
732         orig := line
733
734         // Line is
735         // #cgo [GOOS/GOARCH...] LDFLAGS: stuff

```

```

736 //
737 line = strings.TrimSpace(line)
738 if len(line) < 5 || line[:4] != "#cgo" || (1
739     continue
740 }
741
742 // Split at colon.
743 line = strings.TrimSpace(line[4:])
744 i := strings.Index(line, ":")
745 if i < 0 {
746     return fmt.Errorf("%s: invalid #cgo
747 }
748 line, argstr := line[:i], line[i+1:]
749
750 // Parse GOOS/GOARCH stuff.
751 f := strings.Fields(line)
752 if len(f) < 1 {
753     return fmt.Errorf("%s: invalid #cgo
754 }
755
756 cond, verb := f[:len(f)-1], f[len(f)-1]
757 if len(cond) > 0 {
758     ok := false
759     for _, c := range cond {
760         if ctxt.match(c) {
761             ok = true
762             break
763         }
764     }
765     if !ok {
766         continue
767     }
768 }
769
770 args, err := splitQuoted(argstr)
771 if err != nil {
772     return fmt.Errorf("%s: invalid #cgo
773 }
774 for _, arg := range args {
775     if !safeName(arg) {
776         return fmt.Errorf("%s: malfo
777     }
778 }
779
780 switch verb {
781 case "CFLAGS":
782     di.CgoCFLAGS = append(di.CgoCFLAGS,
783 case "LDFLAGS":
784     di.CgoLDFLAGS = append(di.CgoLDFLAGS

```

```

785         case "pkg-config":
786             di.CgoPkgConfig = append(di.CgoPkgCo
787         default:
788             return fmt.Errorf("%s: invalid #cgo
789         }
790     }
791     return nil
792 }
793
794 var safeBytes = []byte("+-. ,/0123456789=ABCDEFGHIJKLMNQRST
795
796 func safeName(s string) bool {
797     if s == "" {
798         return false
799     }
800     for i := 0; i < len(s); i++ {
801         if c := s[i]; c < 0x80 && bytes.IndexByte(sa
802             return false
803         }
804     }
805     return true
806 }
807
808 // splitQuoted splits the string s around each instance of o
809 // white space characters while taking into account quotes a
810 // returns an array of substrings of s or an empty list if s
811 // Single quotes and double quotes are recognized to prevent
812 // quoted region, and are removed from the resulting substri
813 // isn't closed err will be set and r will have the unclosed
814 // last element. The backslash is used for escaping.
815 //
816 // For example, the following string:
817 //
818 //     a b:"c d" 'e'f' "g\"
819 //
820 // Would be parsed as:
821 //
822 //     []string{"a", "b:c d", "ef", `g`}
823 //
824 func splitQuoted(s string) (r []string, err error) {
825     var args []string
826     arg := make([]rune, len(s))
827     escaped := false
828     quoted := false
829     quote := '\x00'
830     i := 0
831     for _, rune := range s {
832         switch {
833         case escaped:
834             escaped = false

```

```

835         case rune == '\\':
836             escaped = true
837             continue
838         case quote != '\x00':
839             if rune == quote {
840                 quote = '\x00'
841                 continue
842             }
843         case rune == '"' || rune == '\'':
844             quoted = true
845             quote = rune
846             continue
847         case unicode.IsSpace(rune):
848             if quoted || i > 0 {
849                 quoted = false
850                 args = append(args, string(a
851                     i = 0
852             })
853             continue
854     }
855     arg[i] = rune
856     i++
857 }
858 if quoted || i > 0 {
859     args = append(args, string(arg[:i]))
860 }
861 if quote != 0 {
862     err = errors.New("unclosed quote")
863 } else if escaped {
864     err = errors.New("unfinished escaping")
865 }
866 return args, err
867 }
868
869 // match returns true if the name is one of:
870 //
871 //     $GOOS
872 //     $GOARCH
873 //     cgo (if cgo is enabled)
874 //     !cgo (if cgo is disabled)
875 //     tag (if tag is listed in ctxt.BuildTags)
876 //     !tag (if tag is not listed in ctxt.BuildTags)
877 //     a comma-separated list of any of these
878 //
879 func (ctxt *Context) match(name string) bool {
880     if name == "" {
881         return false
882     }
883     if i := strings.Index(name, ","); i >= 0 {

```

```

884         // comma-separated list
885         return ctxt.match(name[:i]) && ctxt.match(name[:i+1])
886     }
887     if strings.HasPrefix(name, "!!") { // bad syntax, re
888         return false
889     }
890     if strings.HasPrefix(name, "!") { // negation
891         return len(name) > 1 && !ctxt.match(name[1:])
892     }
893
894     // Tags must be letters, digits, underscores.
895     // Unlike in Go identifiers, all digits are fine (e.
896     for _, c := range name {
897         if !unicode.IsLetter(c) && !unicode.IsDigit(c) {
898             return false
899         }
900     }
901
902     // special tags
903     if ctxt.CgoEnabled && name == "cgo" {
904         return true
905     }
906     if name == ctxt.GOOS || name == ctxt.GOARCH {
907         return true
908     }
909
910     // other tags
911     for _, tag := range ctxt.BuildTags {
912         if tag == name {
913             return true
914         }
915     }
916
917     return false
918 }
919
920 // goodOSArchFile returns false if the name contains a $GOOS
921 // suffix which does not match the current system.
922 // The recognized name formats are:
923 //
924 //     name_$(GOOS).*
925 //     name_$(GOARCH).*
926 //     name_$(GOOS)_$(GOARCH).*
927 //     name_$(GOOS)_test.*
928 //     name_$(GOARCH)_test.*
929 //     name_$(GOOS)_$(GOARCH)_test.*
930 //
931 func (ctxt *Context) goodOSArchFile(name string) bool {
932     if dot := strings.Index(name, "."); dot != -1 {

```

```

933         name = name[:dot]
934     }
935     l := strings.Split(name, "_")
936     if n := len(l); n > 0 && l[n-1] == "test" {
937         l = l[:n-1]
938     }
939     n := len(l)
940     if n >= 2 && knownOS[l[n-2]] && knownArch[l[n-1]] {
941         return l[n-2] == ctxt.GOOS && l[n-1] == ctxt
942     }
943     if n >= 1 && knownOS[l[n-1]] {
944         return l[n-1] == ctxt.GOOS
945     }
946     if n >= 1 && knownArch[l[n-1]] {
947         return l[n-1] == ctxt.GOARCH
948     }
949     return true
950 }
951
952 var knownOS = make(map[string]bool)
953 var knownArch = make(map[string]bool)
954
955 func init() {
956     for _, v := range strings.Fields(goosList) {
957         knownOS[v] = true
958     }
959     for _, v := range strings.Fields(goarchList) {
960         knownArch[v] = true
961     }
962 }
963
964 // ToolDir is the directory containing build tools.
965 var ToolDir = filepath.Join(runtime.GOROOT(), "pkg/tool/"+ru
966
967 // IsLocalImport reports whether the import path is
968 // a local import path, like ".", "..", "./foo", or "../foo"
969 func IsLocalImport(path string) bool {
970     return path == "." || path == ".." ||
971         strings.HasPrefix(path, "./") || strings.Has
972 }
973
974 // ArchChar returns the architecture character for the given
975 // For example, ArchChar("amd64") returns "6".
976 func ArchChar(goarch string) (string, error) {
977     switch goarch {
978     case "386":
979         return "8", nil
980     case "amd64":
981         return "6", nil
982     case "arm":

```

```
983             return "5", nil
984         }
985         return "", errors.New("unsupported GOARCH " + goarch
986     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/build/doc.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package build gathers information about Go packages.
6 //
7 // Go Path
8 //
9 // The Go path is a list of directory trees containing Go so
10 // It is consulted to resolve imports that cannot be found i
11 // Go tree. The default path is the value of the GOPATH env
12 // variable, interpreted as a path list appropriate to the o
13 // (on Unix, the variable is a colon-separated string;
14 // on Windows, a semicolon-separated string;
15 // on Plan 9, a list).
16 //
17 // Each directory listed in the Go path must have a prescrib
18 //
19 // The src/ directory holds source code. The path below 'sr
20 // the import path or executable name.
21 //
22 // The pkg/ directory holds installed package objects.
23 // As in the Go tree, each target operating system and
24 // architecture pair has its own subdirectory of pkg
25 // (pkg/GOOS_GOARCH).
26 //
27 // If DIR is a directory listed in the Go path, a package wi
28 // source in DIR/src/foo/bar can be imported as "foo/bar" an
29 // has its compiled form installed to "DIR/pkg/GOOS_GOARCH/f
30 // (or, for gccgo, "DIR/pkg/gccgo/foo/libbar.a").
31 //
32 // The bin/ directory holds compiled commands.
33 // Each command is named for its source directory, but only
34 // using the final element, not the entire path. That is, t
35 // command with source in DIR/src/foo/quux is installed into
36 // DIR/bin/quux, not DIR/bin/foo/quux. The foo/ is stripped
37 // so that you can add DIR/bin to your PATH to get at the
38 // installed commands.
39 //
40 // Here's an example directory layout:
41 //
42 //     GOPATH=/home/user/gocode
43 //
44 //     /home/user/gocode/
```

```

45 //          src/
46 //              foo/
47 //                  bar/                (go code in package b
48 //                      x.go
49 //                          quux/        (go code in package m
50 //                              y.go
51 //          bin/
52 //              quux                    (installed command)
53 //          pkg/
54 //              linux_amd64/
55 //                  foo/
56 //                      bar.a            (installed package ob
57 //
58 // Build Constraints
59 //
60 // A build constraint is a line comment beginning with the d
61 // that lists the conditions under which a file should be in
62 // Constraints may appear in any kind of source file (not ju
63 // they must be appear near the top of the file, preceded
64 // only by blank lines and other line comments.
65 //
66 // A build constraint is evaluated as the OR of space-separa
67 // each option evaluates as the AND of its comma-separated t
68 // and each term is an alphanumeric word or, preceded by !,
69 // That is, the build constraint:
70 //
71 //          // +build linux,386 darwin,!cgo
72 //
73 // corresponds to the boolean formula:
74 //
75 //          (linux AND 386) OR (darwin AND (NOT cgo))
76 //
77 // During a particular build, the following words are satisf
78 //
79 //          - the target operating system, as spelled by runtime
80 //          - the target architecture, as spelled by runtime.GOARCH
81 //          - "cgo", if ctxt.CgoEnabled is true
82 //          - any additional words listed in ctxt.BuildTags
83 //
84 // If a file's name, after stripping the extension and a pos
85 // matches *_GOOS, *_GOARCH, or *_GOOS_GOARCH for any known
86 // system and architecture values, then the file is consider
87 // build constraint requiring those terms.
88 //
89 // To keep a file from being considered for the build:
90 //
91 //          // +build ignore
92 //
93 // (any other unsatisfied word will work as well, but ``igno
94 //

```

```
95 // To build a file only when using cgo, and only on Linux an
96 //
97 //      // +build linux,cgo darwin,cgo
98 //
99 // Such a file is usually paired with another file implement
100 // default functionality for other systems, which in this ca
101 // carry the constraint:
102 //
103 //      // +build !linux !darwin !cgo
104 //
105 // Naming a file dns_windows.go will cause it to be included
106 // building the package for Windows; similarly, math_386.s w
107 // only when building the package for 32-bit x86.
108 //
109 package build
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/build/syslist.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package build
6
7 const goosList = "darwin freebsd linux netbsd openbsd plan9
8 const goarchList = "386 amd64 arm "
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/go/doc/comment.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Godoc comment extraction and comment -> HTML formatting.
6
7 package doc
8
9 import (
10     "io"
11     "regexp"
12     "strings"
13     "text/template" // for HTMLEscape
14     "unicode"
15     "unicode/utf8"
16 )
17
18 var (
19     ldquo = []byte("&ldquo;")
20     rdquo = []byte("&rdquo;")
21 )
22
23 // Escape comment text for HTML. If nice is set,
24 // also turn `` into &ldquo; and ' into &rdquo;.
25 func commentEscape(w io.Writer, text string, nice bool) {
26     last := 0
27     if nice {
28         for i := 0; i < len(text)-1; i++ {
29             ch := text[i]
30             if ch == text[i+1] && (ch == '`' ||
31                 template.HTMLEscape(w, []byt
32                 last = i + 2
33                 switch ch {
34                 case '`':
35                     w.Write(ldquo)
36                 case '\':
37                     w.Write(rdquo)
38                 }
39                 i++ // loop will add one mor
40             }
41         }
42     }
```

```

42     }
43     template.HTMLEscape(w, []byte(text[last:]))
44 }
45
46 const (
47     // Regexp for Go identifiers
48     identRx = `[a-zA-Z_][a-zA-Z_0-9]*` // TODO(gri) ASCII
49
50     // Regexp for URLs
51     protocol = `(https?|ftp|file|gopher|mailto|news|nntp
52     hostPart = `[a-zA-Z0-9_@\-]+`
53     filePart = `[a-zA-Z0-9_?%#~&/\-\+=]+`
54     urlRx     = protocol + `//` + // http://
55               hostPart + `([\.:]` + hostPart + `)*/?` + //
56               filePart + `([\.:,]` + filePart + `)*`
57 )
58
59 var matchRx = regexp.MustCompile(`(` + urlRx + `)|(` + ident
60
61 var (
62     html_a      = []byte(`

```

```

92         // write text before match
93         commentEscape(w, line[0:m[0]], nice)
94
95         // analyze match
96         match := line[m[0]:m[1]]
97         url := ""
98         italics := false
99         if words != nil {
100             url, italics = words[string(match)]
101         }
102         if m[2] >= 0 {
103             // match against first parenthesized
104             if !italics {
105                 // no alternative URL in wor
106                 url = string(match)
107             }
108             italics = false // don't italicize U
109         }
110
111         // write match
112         if len(url) > 0 {
113             w.Write(html_a)
114             template.HTMLEscape(w, []byte(url))
115             w.Write(html_aq)
116         }
117         if italics {
118             w.Write(html_i)
119         }
120         commentEscape(w, match, nice)
121         if italics {
122             w.Write(html_endi)
123         }
124         if len(url) > 0 {
125             w.Write(html_enda)
126         }
127
128         // advance
129         line = line[m[1]:]
130     }
131     commentEscape(w, line, nice)
132 }
133
134 func indentLen(s string) int {
135     i := 0
136     for i < len(s) && (s[i] == ' ' || s[i] == '\t') {
137         i++
138     }
139     return i
140 }

```

```

141
142 func isBlank(s string) bool {
143     return len(s) == 0 || (len(s) == 1 && s[0] == '\n')
144 }
145
146 func commonPrefix(a, b string) string {
147     i := 0
148     for i < len(a) && i < len(b) && a[i] == b[i] {
149         i++
150     }
151     return a[0:i]
152 }
153
154 func unindent(block []string) {
155     if len(block) == 0 {
156         return
157     }
158
159     // compute maximum common white prefix
160     prefix := block[0][0:indentLen(block[0])]
161     for _, line := range block {
162         if !isBlank(line) {
163             prefix = commonPrefix(prefix, line[0
164         }
165     }
166     n := len(prefix)
167
168     // remove
169     for i, line := range block {
170         if !isBlank(line) {
171             block[i] = line[n:]
172         }
173     }
174 }
175
176 // heading returns the trimmed line if it passes as a sectio
177 // otherwise it returns the empty string.
178 func heading(line string) string {
179     line = strings.TrimSpace(line)
180     if len(line) == 0 {
181         return ""
182     }
183
184     // a heading must start with an uppercase letter
185     r, _ := utf8.DecodeRuneInString(line)
186     if !unicode.IsLetter(r) || !unicode.IsUpper(r) {
187         return ""
188     }
189

```

```

190         // it must end in a letter or digit:
191         r, _ = utf8.DecodeLastRuneInString(line)
192         if !unicode.IsLetter(r) && !unicode.IsDigit(r) {
193             return ""
194         }
195
196         // exclude lines with illegal characters
197         if strings.IndexAny(line, ",.?!?+*/=())[{}_!^&$~%#@
198             return ""
199     }
200
201     // allow "'" for possessive "'s" only
202     for b := line; ; {
203         i := strings.IndexRune(b, '\'')
204         if i < 0 {
205             break
206         }
207         if i+1 >= len(b) || b[i+1] != 's' || (i+2 <
208             return "" // not followed by "s "
209         }
210         b = b[i+2:]
211     }
212
213     return line
214 }
215
216 type op int
217
218 const (
219     opPara op = iota
220     opHead
221     opPre
222 )
223
224 type block struct {
225     op      op
226     lines []string
227 }
228
229 var nonAlphaNumRx = regexp.MustCompile(`^[a-zA-Z0-9]`)
230
231 func anchorID(line string) string {
232     return nonAlphaNumRx.ReplaceAllString(line, "_")
233 }
234
235 // ToHTML converts comment text to formatted HTML.
236 // The comment was prepared by DocReader,
237 // so it is known not to have leading, trailing blank lines
238 // nor to have trailing spaces at the end of lines.
239 // The comment markers have already been removed.

```

```

240 //
241 // Turn each run of multiple \n into </p><p>.
242 // Turn each run of indented lines into a <pre> block withou
243 // Enclose headings with header tags.
244 //
245 // URLs in the comment text are converted into links; if the
246 // in the words map, the link is taken from the map (if the
247 // value is the empty string, the URL is not converted into
248 //
249 // Go identifiers that appear in the words map are italicize
250 // map value is not the empty string, it is considered a URL
251 // into a link.
252 func ToHTML(w io.Writer, text string, words map[string]strin
253     for _, b := range blocks(text) {
254         switch b.op {
255             case opPara:
256                 w.Write(html_p)
257                 for _, line := range b.lines {
258                     emphasize(w, line, words, tr
259                 }
260                 w.Write(html_endp)
261             case opHead:
262                 w.Write(html_h)
263                 id := ""
264                 for _, line := range b.lines {
265                     if id == "" {
266                         id = anchorID(line)
267                         w.Write([]byte(id))
268                         w.Write(html_hq)
269                     }
270                     commentEscape(w, line, true)
271                 }
272                 if id == "" {
273                     w.Write(html_hq)
274                 }
275                 w.Write(html_endh)
276             case opPre:
277                 w.Write(html_pre)
278                 for _, line := range b.lines {
279                     emphasize(w, line, nil, fals
280                 }
281                 w.Write(html_endpre)
282             }
283         }
284     }
285 }
286 func blocks(text string) []block {
287     var (
288         out []block

```

```

289         para []string
290
291         lastWasBlank = false
292         lastWasHeading = false
293     )
294
295     close := func() {
296         if para != nil {
297             out = append(out, block{opPara, para
298             para = nil
299         }
300     }
301
302     lines := strings.SplitAfter(text, "\n")
303     unindent(lines)
304     for i := 0; i < len(lines); {
305         line := lines[i]
306         if isBlank(line) {
307             // close paragraph
308             close()
309             i++
310             lastWasBlank = true
311             continue
312         }
313         if indentLen(line) > 0 {
314             // close paragraph
315             close()
316
317             // count indented or blank lines
318             j := i + 1
319             for j < len(lines) && (isBlank(lines
320             j++
321         }
322         // but not trailing blank lines
323         for j > i && isBlank(lines[j-1]) {
324             j--
325         }
326         pre := lines[i:j]
327         i = j
328
329         unindent(pre)
330
331         // put those lines in a pre block
332         out = append(out, block{opPre, pre})
333         lastWasHeading = false
334         continue
335     }
336
337     if lastWasBlank && !lastWasHeading && i+2 <

```

```

338         isBlank(lines[i+1]) && !isBlank(line
339         // current line is non-blank, surrou
340         // and the next non-blank line is no
341         // might be a heading.
342         if head := heading(line); head != ""
343             close()
344             out = append(out, block{opHe
345             i += 2
346             lastWasHeading = true
347             continue
348         }
349     }
350
351     // open paragraph
352     lastWasBlank = false
353     lastWasHeading = false
354     para = append(para, lines[i])
355     i++
356 }
357 close()
358
359 return out
360 }
361
362 // ToText prepares comment text for presentation in textual
363 // It wraps paragraphs of text to width or fewer Unicode cod
364 // and then prefixes each line with the indent. In preforma
365 // (such as program text), it prefixes each non-blank line w
366 func ToText(w io.Writer, text string, indent, preIndent stri
367     l := linewriter{
368         out:    w,
369         width:  width,
370         indent: indent,
371     }
372     for _, b := range blocks(text) {
373         switch b.op {
374         case opPara:
375             // l.write will add leading newline
376             for _, line := range b.lines {
377                 l.write(line)
378             }
379             l.flush()
380         case opHead:
381             w.Write(nl)
382             for _, line := range b.lines {
383                 l.write(line + "\n")
384             }
385             l.flush()
386         case opPre:
387             w.Write(nl)

```

```

388         for _, line := range b.lines {
389             if !isBlank(line) {
390                 w.Write([]byte(preIn
391                     w.Write([]byte(line)
392             }
393         }
394     }
395 }
396 }
397
398 type lineWrapper struct {
399     out      io.Writer
400     printed  bool
401     width    int
402     indent   string
403     n        int
404     pendSpace int
405 }
406
407 var nl = []byte("\n")
408 var space = []byte(" ")
409
410 func (l *lineWrapper) write(text string) {
411     if l.n == 0 && l.printed {
412         l.out.Write(nl) // blank line before new par
413     }
414     l.printed = true
415
416     for _, f := range strings.Fields(text) {
417         w := utf8.RuneCountInString(f)
418         // wrap if line is too long
419         if l.n > 0 && l.n+l.pendSpace+w > l.width {
420             l.out.Write(nl)
421             l.n = 0
422             l.pendSpace = 0
423         }
424         if l.n == 0 {
425             l.out.Write([]byte(l.indent))
426         }
427         l.out.Write(space[:l.pendSpace])
428         l.out.Write([]byte(f))
429         l.n += l.pendSpace + w
430         l.pendSpace = 1
431     }
432 }
433
434 func (l *lineWrapper) flush() {
435     if l.n == 0 {
436         return

```

```
437         }
438         l.out.Write(nl)
439         l.pendSpace = 0
440         l.n = 0
441     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/doc/doc.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package doc extracts source code documentation from a Go
6 package doc
7
8 import (
9     "go/ast"
10    "go/token"
11 )
12
13 // Package is the documentation for an entire package.
14 type Package struct {
15     Doc          string
16     Name         string
17     ImportPath   string
18     Imports      []string
19     Filenames    []string
20     Bugs         []string
21
22     // declarations
23     Consts []*Value
24     Types  []*Type
25     Vars   []*Value
26     Funcs  []*Func
27 }
28
29 // Value is the documentation for a (possibly grouped) var or c
30 type Value struct {
31     Doc      string
32     Names    []string // var or const names in declaration
33     Decl     *ast.GenDecl
34
35     order int
36 }
37
38 // Type is the documentation for a type declaration.
39 type Type struct {
40     Doc      string
41     Name     string
42     Decl     *ast.GenDecl
43
44     // associated declarations
```

```

45         Consts  []*Value // sorted list of constants of (mos
46         Vars    []*Value // sorted list of variables of (mos
47         Funcs   []*Func  // sorted list of functions returni
48         Methods []*Func  // sorted list of methods (includin
49     }
50
51 // Func is the documentation for a func declaration.
52 type Func struct {
53     Doc string
54     Name string
55     Decl *ast.FuncDecl
56
57     // methods
58     // (for functions, these fields have the respective
59     Recv string // actual receiver "T" or "*T"
60     Orig string // original receiver "T" or "*T"
61     Level int   // embedding level; 0 means not embedde
62 }
63
64 // Mode values control the operation of New.
65 type Mode int
66
67 const (
68     // extract documentation for all package-level decla
69     // not just exported ones
70     AllDecls Mode = 1 << iota
71
72     // show all embedded methods, not just the ones of
73     // invisible (unexported) anonymous fields
74     AllMethods
75 )
76
77 // New computes the package documentation for the given pack
78 // New takes ownership of the AST pkg and may edit or overwr
79 //
80 func New(pkg *ast.Package, importPath string, mode Mode) *Pa
81     var r reader
82     r.readPackage(pkg, mode)
83     r.computeMethodSets()
84     r.cleanupTypes()
85     return &Package{
86         Doc:      r.doc,
87         Name:      pkg.Name,
88         ImportPath: importPath,
89         Imports:   sortedKeys(r.imports),
90         Filenames: r.filenames,
91         Bugs:      r.bugs,
92         Consts:    sortedValues(r.values, token.CON
93         Types:    sortedTypes(r.types, mode&AllMet
94         Vars:      sortedValues(r.values, token.VAR

```

```
95         Funcs:         sortedFuncs(r.funcs, true),
96     }
97 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/doc/example.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Extract example functions from file ASTs.
6
7 package doc
8
9 import (
10     "go/ast"
11     "go/token"
12     "regexp"
13     "sort"
14     "strings"
15     "unicode"
16     "unicode/utf8"
17 )
18
19 type Example struct {
20     Name      string // name of the item being exemplified
21     Doc       string // example function doc string
22     Code      ast.Node
23     Comments  []*ast.CommentGroup
24     Output    string // expected output
25 }
26
27 func Examples(files ...*ast.File) []*Example {
28     var list []*Example
29     for _, file := range files {
30         hasTests := false // file contains tests or
31         numDecl := 0      // number of non-import declarations
32         var flist []*Example
33         for _, decl := range file.Decls {
34             if g, ok := decl.(*ast.GenDecl); ok {
35                 numDecl++
36                 continue
37             }
38             f, ok := decl.(*ast.FuncDecl)
39             if !ok {
40                 continue
41             }
42             numDecl++
43             name := f.Name.Name
44             if isTest(name, "Test") || isTest(name,
```

```

45             hasTests = true
46             continue
47         }
48         if !isTest(name, "Example") {
49             continue
50         }
51         var doc string
52         if f.Doc != nil {
53             doc = f.Doc.Text()
54         }
55         flist = append(flist, &Example{
56             Name:     name[len("Example")
57             Doc:      doc,
58             Code:     f.Body,
59             Comments: file.Comments,
60             Output:   exampleOutput(f, f
61         })
62     }
63     if !hasTests && numDecl > 1 && len(flist) ==
64         // If this file only has one example
65         // other top-level declarations, and
66         // benchmarks, use the whole file as
67         flist[0].Code = file
68     }
69     list = append(list, flist...)
70 }
71 sort.Sort(exampleByName(list))
72 return list
73 }
74
75 var outputPrefix = regexp.MustCompile(`(?i)^[[:space:]]*outp
76
77 func exampleOutput(fun *ast.FuncDecl, comments []*ast.Commen
78     // find the last comment in the function
79     var last *ast.CommentGroup
80     for _, cg := range comments {
81         if cg.Pos() < fun.Pos() {
82             continue
83         }
84         if cg.End() > fun.End() {
85             break
86         }
87         last = cg
88     }
89     if last != nil {
90         // test that it begins with the correct pref
91         text := last.Text()
92         if loc := outputPrefix.FindStringIndex(text)
93             return strings.TrimSpace(text[loc[1]
94     }

```

```

95         }
96         return "" // no suitable comment found
97     }
98
99     // isTest tells whether name looks like a test, example, or
100    // It is a Test (say) if there is a character after Test tha
101    // lower-case letter. (We don't want Testiness.)
102    func isTest(name, prefix string) bool {
103        if !strings.HasPrefix(name, prefix) {
104            return false
105        }
106        if len(name) == len(prefix) { // "Test" is ok
107            return true
108        }
109        rune, _ := utf8.DecodeRuneInString(name[len(prefix):
110        return !unicode.IsLower(rune)
111    }
112
113    type exampleByName []*Example
114
115    func (s exampleByName) Len() int           { return len(s) }
116    func (s exampleByName) Swap(i, j int)      { s[i], s[j] = s[
117    func (s exampleByName) Less(i, j int) bool { return s[i].Name

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/doc/exports.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file implements export filtering of an AST.
6
7 package doc
8
9 import "go/ast"
10
11 // filterIdentList removes unexported names from list in pla
12 // and returns the resulting list.
13 //
14 func filterIdentList(list []*ast.Ident) []*ast.Ident {
15     j := 0
16     for _, x := range list {
17         if ast.IsExported(x.Name) {
18             list[j] = x
19             j++
20         }
21     }
22     return list[0:j]
23 }
24
25 // removeErrorField removes anonymous fields named "error" f
26 // This is called when "error" has been determined to be a l
27 // not the predeclared type.
28 //
29 func removeErrorField(ityp *ast.InterfaceType) {
30     list := ityp.Methods.List // we know that ityp.Metho
31     j := 0
32     for _, field := range list {
33         keepField := true
34         if n := len(field.Names); n == 0 {
35             // anonymous field
36             if fname, _ := baseTypeName(field.Ty
37                 keepField = false
38         }
39     }
40     if keepField {
41         list[j] = field
42         j++
43     }
44 }
```

```

45     if j < len(list) {
46         ityp.Incomplete = true
47     }
48     ityp.Methods.List = list[0:j]
49 }
50
51 // filterFieldList removes unexported fields (field names) f
52 // in place and returns true if fields were removed. Anonymo
53 // recorded with the parent type. filterType is called with
54 // all remaining fields.
55 //
56 func (r *reader) filterFieldList(parent *namedType, fields *
57     if fields == nil {
58         return
59     }
60     list := fields.List
61     j := 0
62     for _, field := range list {
63         keepField := false
64         if n := len(field.Names); n == 0 {
65             // anonymous field
66             fname := r.recordAnonymousField(pare
67             if ast.IsExported(fname) {
68                 keepField = true
69             } else if ityp != nil && fname == "e
70                 // possibly the predeclared
71                 // it for now but remember t
72                 // it can be fixed if error
73                 keepField = true
74                 r.remember(ityp)
75             }
76         } else {
77             field.Names = filterIdentList(field.
78             if len(field.Names) < n {
79                 removedFields = true
80             }
81             if len(field.Names) > 0 {
82                 keepField = true
83             }
84         }
85         if keepField {
86             r.filterType(nil, field.Type)
87             list[j] = field
88             j++
89         }
90     }
91     if j < len(list) {
92         removedFields = true
93     }
94     fields.List = list[0:j]

```

```

95         return
96     }
97
98     // filterParamList applies filterType to each parameter type
99     //
100    func (r *reader) filterParamList(fields *ast.FieldList) {
101        if fields != nil {
102            for _, f := range fields.List {
103                r.filterType(nil, f.Type)
104            }
105        }
106    }
107
108    // filterType strips any unexported struct fields or method
109    // in place. If fields (or methods) have been removed, the c
110    // struct or interface type has the Incomplete field set to
111    //
112    func (r *reader) filterType(parent *namedType, typ ast.Expr)
113        switch t := typ.(type) {
114        case *ast.Ident:
115            // nothing to do
116        case *ast.ParenExpr:
117            r.filterType(nil, t.X)
118        case *ast.ArrayType:
119            r.filterType(nil, t.Elt)
120        case *ast.StructType:
121            if r.filterFieldList(parent, t.Fields, nil)
122                t.Incomplete = true
123            }
124        case *ast.FuncType:
125            r.filterParamList(t.Params)
126            r.filterParamList(t.Results)
127        case *ast.InterfaceType:
128            if r.filterFieldList(parent, t.Methods, t) {
129                t.Incomplete = true
130            }
131        case *ast.MapType:
132            r.filterType(nil, t.Key)
133            r.filterType(nil, t.Value)
134        case *ast.ChanType:
135            r.filterType(nil, t.Value)
136        }
137    }
138
139    func (r *reader) filterSpec(spec ast.Spec) bool {
140        switch s := spec.(type) {
141        case *ast.ImportSpec:
142            // always keep imports so we can collect the
143            return true

```

```

144     case *ast.ValueSpec:
145         s.Names = filterIdentList(s.Names)
146         if len(s.Names) > 0 {
147             r.filterType(nil, s.Type)
148             return true
149         }
150     case *ast.TypeSpec:
151         if name := s.Name.Name; ast.IsExported(name)
152             r.filterType(r.lookupType(s.Name.Name)
153             return true
154         } else if name == "error" {
155             // special case: remember that error
156             r.errorDecl = true
157         }
158     }
159     return false
160 }
161
162 func (r *reader) filterSpecList(list []ast.Spec) []ast.Spec
163     j := 0
164     for _, s := range list {
165         if r.filterSpec(s) {
166             list[j] = s
167             j++
168         }
169     }
170     return list[0:j]
171 }
172
173 func (r *reader) filterDecl(decl ast.Decl) bool {
174     switch d := decl.(type) {
175     case *ast.GenDecl:
176         d.Specs = r.filterSpecList(d.Specs)
177         return len(d.Specs) > 0
178     case *ast.FuncDecl:
179         // ok to filter these methods early because
180         // conflicting method will be filtered here,
181         // thus, removing these methods early will n
182         // to the false removal of possible conflict
183         return ast.IsExported(d.Name.Name)
184     }
185     return false
186 }
187
188 // fileExports removes unexported declarations from src in p
189 //
190 func (r *reader) fileExports(src *ast.File) {
191     j := 0
192     for _, d := range src.Decls {

```

```
193             if r.filterDecl(d) {
194                 src.Decls[j] = d
195                 j++
196             }
197         }
198         src.Decls = src.Decls[0:j]
199     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/doc/filter.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package doc
6
7 import "go/ast"
8
9 type Filter func(string) bool
10
11 func matchFields(fields *ast.FieldList, f Filter) bool {
12     if fields != nil {
13         for _, field := range fields.List {
14             for _, name := range field.Names {
15                 if f(name.Name) {
16                     return true
17                 }
18             }
19         }
20     }
21     return false
22 }
23
24 func matchDecl(d *ast.GenDecl, f Filter) bool {
25     for _, d := range d.Specs {
26         switch v := d.(type) {
27         case *ast.ValueSpec:
28             for _, name := range v.Names {
29                 if f(name.Name) {
30                     return true
31                 }
32             }
33         case *ast.TypeSpec:
34             if f(v.Name.Name) {
35                 return true
36             }
37             switch t := v.Type.(type) {
38             case *ast.StructType:
39                 if matchFields(t.Fields, f) {
40                     return true
41                 }
42             case *ast.InterfaceType:
43                 if matchFields(t.Methods, f) {
44                     return true
```

```

45         }
46     }
47 }
48 }
49     return false
50 }
51
52 func filterValues(a []*Value, f Filter) []*Value {
53     w := 0
54     for _, vd := range a {
55         if matchDecl(vd.Decl, f) {
56             a[w] = vd
57             w++
58         }
59     }
60     return a[0:w]
61 }
62
63 func filterFuncs(a []*Func, f Filter) []*Func {
64     w := 0
65     for _, fd := range a {
66         if f(fd.Name) {
67             a[w] = fd
68             w++
69         }
70     }
71     return a[0:w]
72 }
73
74 func filterTypes(a []*Type, f Filter) []*Type {
75     w := 0
76     for _, td := range a {
77         n := 0 // number of matches
78         if matchDecl(td.Decl, f) {
79             n = 1
80         } else {
81             // type name doesn't match, but we r
82             td.Consts = filterValues(td.Consts,
83             td.Vars = filterValues(td.Vars, f)
84             td.Funcs = filterFuncs(td.Funcs, f)
85             td.Methods = filterFuncs(td.Methods,
86             n += len(td.Consts) + len(td.Vars) +
87         }
88         if n > 0 {
89             a[w] = td
90             w++
91         }
92     }
93     return a[0:w]
94 }

```

```
95
96 // Filter eliminates documentation for names that don't pass
97 // TODO: Recognize "Type.Method" as a name.
98 //
99 func (p *Package) Filter(f Filter) {
100     p.Consts = filterValues(p.Consts, f)
101     p.Vars = filterValues(p.Vars, f)
102     p.Types = filterTypes(p.Types, f)
103     p.Funcs = filterFuncs(p.Funcs, f)
104     p.Doc = "" // don't show top-level package doc
105 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/doc/reader.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package doc
6
7 import (
8     "go/ast"
9     "go/token"
10    "regexp"
11    "sort"
12    "strconv"
13 )
14
15 // -----
16 // function/method sets
17 //
18 // Internally, we treat functions like methods and collect t
19
20 // A methodSet describes a set of methods. Entries where Dec
21 // entries (more then one method with the same name at the s
22 //
23 type methodSet map[string]*Func
24
25 // recvString returns a string representation of recv of the
26 // form "T", "*T", or "BADRECV" (if not a proper receiver ty
27 //
28 func recvString(recv ast.Expr) string {
29     switch t := recv.(type) {
30     case *ast.Ident:
31         return t.Name
32     case *ast.StarExpr:
33         return "*" + recvString(t.X)
34     }
35     return "BADRECV"
36 }
37
38 // set creates the corresponding Func for f and adds it to m
39 // If there are multiple f's with the same name, set keeps t
40 // one with documentation; conflicts are ignored.
41 //
42 func (mset methodSet) set(f *ast.FuncDecl) {
43     name := f.Name.Name
44     if g := mset[name]; g != nil && g.Doc != "" {
```

```

45         // A function with the same name has already
46         // since it has documentation, assume f is s
47         // implementation and ignore it. This does n
48         // caller is using go/build.ScanDir to deter
49         // files implementing a package.
50         return
51     }
52     // function doesn't exist or has no documentation; u
53     recv := ""
54     if f.Recv != nil {
55         var typ ast.Expr
56         // be careful in case of incorrect ASTs
57         if list := f.Recv.List; len(list) == 1 {
58             typ = list[0].Type
59         }
60         recv = recvString(typ)
61     }
62     mset[name] = &Func{
63         Doc:  f.Doc.Text(),
64         Name: name,
65         Decl: f,
66         Recv: recv,
67         Orig: recv,
68     }
69     f.Doc = nil // doc consumed - remove from AST
70 }
71
72 // add adds method m to the method set; m is ignored if the
73 // already contains a method with the same name at the same
74 // level then m.
75 //
76 func (mset methodSet) add(m *Func) {
77     old := mset[m.Name]
78     if old == nil || m.Level < old.Level {
79         mset[m.Name] = m
80         return
81     }
82     if old != nil && m.Level == old.Level {
83         // conflict - mark it using a method with ni
84         mset[m.Name] = &Func{
85             Name:  m.Name,
86             Level: m.Level,
87         }
88     }
89 }
90
91 // -----
92 // Named types
93
94 // baseTypeName returns the name of the base type of x (or "

```

```

95 // and whether the type is imported or not.
96 //
97 func baseTypeName(x ast.Expr) (name string, imported bool) {
98     switch t := x.(type) {
99     case *ast.Ident:
100         return t.Name, false
101     case *ast.SelectorExpr:
102         if _, ok := t.X.(*ast.Ident); ok {
103             // only possible for qualified type
104             // assume type is imported
105             return t.Sel.Name, true
106         }
107     case *ast.StarExpr:
108         return baseTypeName(t.X)
109     }
110     return
111 }
112
113 // An embeddedSet describes a set of embedded types.
114 type embeddedSet map[*namedType]bool
115
116 // A namedType represents a named unqualified (package local
117 // predeclared) type. The namedType for a type name is always
118 // reader.lookupType.
119 //
120 type namedType struct {
121     doc string // doc comment for type
122     name string // type name
123     decl *ast.GenDecl // nil if declaration hasn't been
124
125     isEmbedded bool // true if this type is embed
126     isStruct bool // true if this type is a str
127     embedded embeddedSet // true if the embedded type
128
129     // associated declarations
130     values []*Value // consts and vars
131     funcs methodSet
132     methods methodSet
133 }
134
135 // -----
136 // AST reader
137
138 // reader accumulates documentation for a single package.
139 // It modifies the AST: Comments (declaration documentation)
140 // that have been collected by the reader are set to nil
141 // in the respective AST nodes so that they are not printed
142 // twice (once when printing the documentation and once when
143 // printing the corresponding AST node).

```

```

144 //
145 type reader struct {
146     mode Mode
147
148     // package properties
149     doc      string // package documentation, if any
150     filenames []string
151     bugs      []string
152
153     // declarations
154     imports map[string]int
155     values  []*Value // consts and vars
156     types   map[string]*namedType
157     funcs   methodSet
158
159     // support for package-local error type declarations
160     errorDecl bool // if set, type "erro
161     fixlist  []*ast.InterfaceType // list of interfaces
162 }
163
164 func (r *reader) isVisible(name string) bool {
165     return r.mode&AllDecls != 0 || ast.IsExported(name)
166 }
167
168 // lookupType returns the base type with the given name.
169 // If the base type has not been encountered yet, a new
170 // type with the given name but no associated declaration
171 // is added to the type map.
172 //
173 func (r *reader) lookupType(name string) *namedType {
174     if name == "" || name == "_" {
175         return nil // no type docs for anonymous typ
176     }
177     if typ, found := r.types[name]; found {
178         return typ
179     }
180     // type not found - add one without declaration
181     typ := &namedType{
182         name:      name,
183         embedded:  make(embeddedSet),
184         funcs:     make(methodSet),
185         methods:  make(methodSet),
186     }
187     r.types[name] = typ
188     return typ
189 }
190
191 // recordAnonymousField registers fieldType as the type of a
192 // anonymous field in the parent type. If the field is impor

```

```

193 // (qualified name) or the parent is nil, the field is ignor
194 // The function returns the field name.
195 //
196 func (r *reader) recordAnonymousField(parent *namedType, fie
197     fname, imp := baseTypeName(fieldType)
198     if parent == nil || imp {
199         return
200     }
201     if ftype := r.lookupType(fname); ftype != nil {
202         ftype.isEmbedded = true
203         _, ptr := fieldType.(*ast.StarExpr)
204         parent.embedded[ftype] = ptr
205     }
206     return
207 }
208
209 func (r *reader) readDoc(comment *ast.CommentGroup) {
210     // By convention there should be only one package co
211     // but collect all of them if there are more then on
212     text := comment.Text()
213     if r.doc == "" {
214         r.doc = text
215         return
216     }
217     r.doc += "\n" + text
218 }
219
220 func (r *reader) remember(typ *ast.InterfaceType) {
221     r.fixlist = append(r.fixlist, typ)
222 }
223
224 func specNames(specs []ast.Spec) []string {
225     names := make([]string, 0, len(specs)) // reasonable
226     for _, s := range specs {
227         // s guaranteed to be an *ast.ValueSpec by r
228         for _, ident := range s.(*ast.ValueSpec).Name
229             names = append(names, ident.Name)
230     }
231 }
232     return names
233 }
234
235 // readValue processes a const or var declaration.
236 //
237 func (r *reader) readValue(decl *ast.GenDecl) {
238     // determine if decl should be associated with a typ
239     // Heuristic: For each typed entry, determine the ty
240     //             If there is exactly one type name that
241     //             frequent, associate the decl with the
242     domName := ""

```

```

243     domFreq := 0
244     prev := ""
245     n := 0
246     for _, spec := range decl.Specs {
247         s, ok := spec.(*ast.ValueSpec)
248         if !ok {
249             continue // should not happen, but b
250         }
251         name := ""
252         switch {
253         case s.Type != nil:
254             // a type is present; determine its
255             if n, imp := baseTypeName(s.Type); !
256                 name = n
257         }
258         case decl.Tok == token.CONST:
259             // no type is present but we have a
260             // use the previous type name (w/o r
261             // we cannot handle the case of unna
262             // initializer expressions except fo
263             name = prev
264         }
265         if name != "" {
266             // entry has a named type
267             if domName != "" && domName != name
268                 // more than one type name -
269                 // with any type
270                 domName = ""
271                 break
272             }
273             domName = name
274             domFreq++
275         }
276         prev = name
277         n++
278     }
279
280     // nothing to do w/o a legal declaration
281     if n == 0 {
282         return
283     }
284
285     // determine values list with which to associate the
286     values := &r.values
287     const threshold = 0.75
288     if domName != "" && r.isVisible(domName) && domFreq
289         // typed entries are sufficiently frequent
290         if typ := r.lookupType(domName); typ != nil
291             values = &typ.values // associate wi

```

```

292         }
293     }
294
295     *values = append(*values, &Value{
296         Doc:    decl.Doc.Text(),
297         Names: specNames(decl.Specs),
298         Decl:   decl,
299         order: len(*values),
300     })
301     decl.Doc = nil // doc consumed - remove from AST
302 }
303
304 // fields returns a struct's fields or an interface's method
305 //
306 func fields(typ ast.Expr) (list []*ast.Field, isStruct bool)
307     var fields *ast.FieldList
308     switch t := typ.(type) {
309     case *ast.StructType:
310         fields = t.Fields
311         isStruct = true
312     case *ast.InterfaceType:
313         fields = t.Methods
314     }
315     if fields != nil {
316         list = fields.List
317     }
318     return
319 }
320
321 // readType processes a type declaration.
322 //
323 func (r *reader) readType(decl *ast.GenDecl, spec *ast.TypeS
324     typ := r.lookupType(spec.Name.Name)
325     if typ == nil {
326         return // no name or blank name - ignore the
327     }
328
329     // A type should be added at most once, so typ.decl
330     // should be nil - if it is not, simply overwrite it
331     typ.decl = decl
332
333     // compute documentation
334     doc := spec.Doc
335     spec.Doc = nil // doc consumed - remove from AST
336     if doc == nil {
337         // no doc associated with the spec, use the
338         doc = decl.Doc
339     }
340     decl.Doc = nil // doc consumed - remove from AST

```

```

341     typ.doc = doc.Text()
342
343     // record anonymous fields (they may contribute meth
344     // (some fields may have been recorded already when
345     // exports, but that's ok)
346     var list []*ast.Field
347     list, typ.isStruct = fields(spec.Type)
348     for _, field := range list {
349         if len(field.Names) == 0 {
350             r.recordAnonymousField(typ, field.Ty
351         }
352     }
353 }
354
355 // readFunc processes a func or method declaration.
356 //
357 func (r *reader) readFunc(fun *ast.FuncDecl) {
358     // strip function body
359     fun.Body = nil
360
361     // associate methods with the receiver type, if any
362     if fun.Recv != nil {
363         // method
364         recvTypeName, imp := baseTypeName(fun.Recv.L
365         if imp {
366             // should not happen (incorrect AST)
367             // don't show this method
368             return
369         }
370         if typ := r.lookupType(recvTypeName); typ !=
371             typ.methods.set(fun)
372     }
373     // otherwise ignore the method
374     // TODO(gri): There may be exported methods
375     // that can be called because of exported va
376     // function results) of that type. Could det
377     // case and then show those methods in an ap
378     return
379 }
380
381 // associate factory functions with the first visibl
382 if fun.Type.Results.NumFields() >= 1 {
383     res := fun.Type.Results.List[0]
384     if len(res.Names) <= 1 {
385         // exactly one (named or anonymous)
386         // with the first type in result sig
387         // be more than one result)
388         if n, imp := baseTypeName(res.Type);
389             if typ := r.lookupType(n); t
390             // associate functio

```

```

391                                     typ.funcs.set(fun)
392                                     return
393                                     }
394                                 }
395                             }
396                         }
397
398                     // just an ordinary function
399                     r.funcs.set(fun)
400                 }
401
402     var (
403         bug_markers = regexp.MustCompile("^/[/*][ \t]*BUG\\\(
404         bug_content = regexp.MustCompile("[^ \n\r\t]+")
405     )
406
407     // readFile adds the AST for a source file to the reader.
408     //
409     func (r *reader) readFile(src *ast.File) {
410         // add package documentation
411         if src.Doc != nil {
412             r.readDoc(src.Doc)
413             src.Doc = nil // doc consumed - remove from
414         }
415
416         // add all declarations
417         for _, decl := range src.Decls {
418             switch d := decl.(type) {
419             case *ast.GenDecl:
420                 switch d.Tok {
421                 case token.IMPORT:
422                     // imports are handled indiv
423                     for _, spec := range d.Specs
424                     if s, ok := spec.(*a
425                     if import_,
426                     r.im
427                 }
428             }
429         }
430         case token.CONST, token.VAR:
431             // constants and variables a
432             r.readValue(d)
433         case token.TYPE:
434             // types are handled individ
435             if len(d.Specs) == 1 && !d.L
436                 // common case: sing
437                 // (if a single decl
438                 // create a new fake
439                 // go/doc type decla

```

```

440             // parentheses)
441             if s, ok := d.Specs[
442                 r.readType(d
443             }
444             break
445         }
446         for _, spec := range d.Specs
447             if s, ok := spec.(*a
448                 // use an in
449                 // for each
450                 // gets to (
451                 // if there'
452                 fake := &ast
453                     Doc:
454                     // d
455                     // w
456                     // t
457                     // t
458                     // s
459                     TokP
460                     Tok:
461                     Spec
462             }
463             r.readType(f
464         }
465     }
466     }
467     case *ast.FuncDecl:
468         r.readFunc(d)
469     }
470 }
471
472 // collect BUG(...) comments
473 for _, c := range src.Comments {
474     text := c.List[0].Text
475     if m := bug_markers.FindStringIndex(text); m
476         // found a BUG comment; maybe empty
477         if btxt := text[m[1]:]; bug_content.
478             // non-empty BUG comment; co
479             list := append([]*ast.Commen
480             list[0].Text = text[m[1]:]
481             r.bugs = append(r.bugs, (&as
482         }
483     }
484 }
485 src.Comments = nil // consumed unassociated comments
486 }
487
488 func (r *reader) readPackage(pkg *ast.Package, mode Mode) {

```

```

489         // initialize reader
490         r.filesnames = make([]string, len(pkg.Files))
491         r.imports = make(map[string]int)
492         r.mode = mode
493         r.types = make(map[string]*namedType)
494         r.funcs = make(methodSet)
495
496         // sort package files before reading them so that th
497         // result result does not depend on map iteration or
498         i := 0
499         for filename := range pkg.Files {
500             r.filesnames[i] = filename
501             i++
502         }
503         sort.Strings(r.filesnames)
504
505         // process files in sorted order
506         for _, filename := range r.filesnames {
507             f := pkg.Files[filename]
508             if mode&AllDecls == 0 {
509                 r.fileExports(f)
510             }
511             r.readFile(f)
512         }
513     }
514
515     // -----
516     // Types
517
518     var predeclaredTypes = map[string]bool{
519         "bool":      true,
520         "byte":      true,
521         "complex64": true,
522         "complex128": true,
523         "error":     true,
524         "float32":   true,
525         "float64":   true,
526         "int":       true,
527         "int8":      true,
528         "int16":     true,
529         "int32":     true,
530         "int64":     true,
531         "rune":     true,
532         "string":    true,
533         "uint":     true,
534         "uint8":    true,
535         "uint16":   true,
536         "uint32":   true,
537         "uint64":   true,
538         "uintptr":  true,

```

```

539 }
540
541 func customizeRecv(f *Func, recvTypeName string, embeddedIsP
542     if f == nil || f.Decl == nil || f.Decl.Recv == nil |
543         return f // shouldn't happen, but be safe
544     }
545
546     // copy existing receiver field and set new type
547     newField := *f.Decl.Recv.List[0]
548     _, origRecvIsPtr := newField.Type.(*ast.StarExpr)
549     var typ ast.Expr = ast.NewIdent(recvTypeName)
550     if !embeddedIsPtr && origRecvIsPtr {
551         typ = &ast.StarExpr{X: typ}
552     }
553     newField.Type = typ
554
555     // copy existing receiver field list and set new rec
556     newFieldList := *f.Decl.Recv
557     newFieldList.List = []*ast.Field{&newField}
558
559     // copy existing function declaration and set new re
560     newFuncDecl := *f.Decl
561     newFuncDecl.Recv = &newFieldList
562
563     // copy existing function documentation and set new
564     newF := *f
565     newF.Decl = &newFuncDecl
566     newF.Recv = recvString(typ)
567     // the Orig field never changes
568     newF.Level = level
569
570     return &newF
571 }
572
573 // collectEmbeddedMethods collects the embedded methods of t
574 //
575 func (r *reader) collectEmbeddedMethods(mset methodSet, typ
576     visited[typ] = true
577     for embedded, isPtr := range typ.embedded {
578         // Once an embedded type is embedded as a po
579         // all embedded types in those types are tre
580         // pointer types for the purpose of the rece
581         // computation; i.e., embeddedIsPtr is stick
582         // embedding hierarchy.
583         thisEmbeddedIsPtr := embeddedIsPtr || isPtr
584         for _, m := range embedded.methods {
585             // only top-level methods are embedd
586             if m.Level == 0 {
587                 mset.add(customizeRecv(m, re

```

```

588         }
589     }
590     if !visited[embedded] {
591         r.collectEmbeddedMethods(mset, embed
592     }
593 }
594 delete(visited, typ)
595 }
596
597 // computeMethodSets determines the actual method sets for e
598 //
599 func (r *reader) computeMethodSets() {
600     for _, t := range r.types {
601         // collect embedded methods for t
602         if t.isStruct {
603             // struct
604             r.collectEmbeddedMethods(t.methods,
605         } else {
606             // interface
607             // TODO(gri) fix this
608         }
609     }
610
611     // if error was declared locally, don't treat it as
612     if r.errorDecl {
613         for _, ityp := range r.fixlist {
614             removeErrorField(ityp)
615         }
616     }
617 }
618
619 // cleanupTypes removes the association of functions and met
620 // types that have no declaration. Instead, these functions
621 // are shown at the package level. It also removes types wit
622 // declarations or which are not visible.
623 //
624 func (r *reader) cleanupTypes() {
625     for _, t := range r.types {
626         visible := r.isVisible(t.name)
627         if t.decl == nil && (predeclaredTypes[t.name
628             // t.name is a predeclared type (and
629             // or it was embedded somewhere but
630             // the AST is incomplete): move any
631             // back to the top-level so that the
632             // 1) move values
633             r.values = append(r.values, t.values
634             // 2) move factory functions
635             for name, f := range t.funcs {
636                 // in a correct AST, package

```

```

637             // are all different - no ne
638             r.funcs[name] = f
639         }
640         // 3) move methods
641         for name, m := range t.methods {
642             // don't overwrite functions
643             if _, found := r.funcs[name]
644                 r.funcs[name] = m
645             }
646         }
647     }
648     // remove types w/o declaration or which are
649     if t.decl == nil || !visible {
650         delete(r.types, t.name)
651     }
652 }
653 }
654
655 // -----
656 // Sorting
657
658 type data struct {
659     n      int
660     swap  func(i, j int)
661     less  func(i, j int) bool
662 }
663
664 func (d *data) Len() int           { return d.n }
665 func (d *data) Swap(i, j int)     { d.swap(i, j) }
666 func (d *data) Less(i, j int) bool { return d.less(i, j) }
667
668 // sortBy is a helper function for sorting
669 func sortBy(less func(i, j int) bool, swap func(i, j int), n
670     sort.Sort(&data{n, swap, less})
671 }
672
673 func sortedKeys(m map[string]int) []string {
674     list := make([]string, len(m))
675     i := 0
676     for key := range m {
677         list[i] = key
678         i++
679     }
680     sort.Strings(list)
681     return list
682 }
683
684 // sortingName returns the name to use when sorting d into p
685 //
686 func sortingName(d *ast.GenDecl) string {

```

```

687         if len(d.Specs) == 1 {
688             if s, ok := d.Specs[0].(*ast.ValueSpec); ok
689                 return s.Names[0].Name
690         }
691     }
692     return ""
693 }
694
695 func sortedValues(m []*Value, tok token.Token) []*Value {
696     list := make([]*Value, len(m)) // big enough in any
697     i := 0
698     for _, val := range m {
699         if val.Decl.Tok == tok {
700             list[i] = val
701             i++
702         }
703     }
704     list = list[0:i]
705
706     sortBy(
707         func(i, j int) bool {
708             if ni, nj := sortingName(list[i].Decl
709                 return ni < nj
710             }
711             return list[i].order < list[j].order
712         },
713         func(i, j int) { list[i], list[j] = list[j],
714             len(list),
715     )
716
717     return list
718 }
719
720 func sortedTypes(m map[string]*namedType, allMethods bool) [
721     list := make([]*Type, len(m))
722     i := 0
723     for _, t := range m {
724         list[i] = &Type{
725             Doc:    t.doc,
726             Name:    t.name,
727             Decl:    t.decl,
728             Consts: sortedValues(t.values, toke
729             Vars:   sortedValues(t.values, toke
730             Funcs: sortedFuncs(t.funcs, true),
731             Methods: sortedFuncs(t.methods, allM
732         }
733         i++
734     }
735

```

```

736         sortBy(
737             func(i, j int) bool { return list[i].Name <
738                 func(i, j int) { list[i], list[j] = list[j],
739                     len(list),
740                 )
741         }
742         return list
743     }
744
745     func removeStar(s string) string {
746         if len(s) > 0 && s[0] == '*' {
747             return s[1:]
748         }
749         return s
750     }
751
752     func sortedFuncs(m methodSet, allMethods bool) []*Func {
753         list := make([]*Func, len(m))
754         i := 0
755         for _, m := range m {
756             // determine which methods to include
757             switch {
758             case m.Decl == nil:
759                 // exclude conflict entry
760             case allMethods, m.Level == 0, !ast.IsExport:
761                 // forced inclusion, method not embe
762                 // embedded but original receiver ty
763                 list[i] = m
764                 i++
765             }
766         }
767         list = list[0:i]
768         sortBy(
769             func(i, j int) bool { return list[i].Name <
770                 func(i, j int) { list[i], list[j] = list[j],
771                     len(list),
772                 )
773         }
774         return list
775     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/doc/synopsis.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package doc
6
7 import "unicode"
8
9 // firstSentenceLen returns the length of the first sentence
10 // The sentence ends after the first period followed by spac
11 // not preceded by exactly one uppercase letter.
12 //
13 func firstSentenceLen(s string) int {
14     var ppp, pp, p rune
15     for i, q := range s {
16         if q == '\n' || q == '\r' || q == '\t' {
17             q = ' '
18         }
19         if q == ' ' && p == '.' && (!unicode.IsUpper
20             return i
21         }
22         ppp, pp, p = pp, p, q
23     }
24     return len(s)
25 }
26
27 // Synopsis returns a cleaned version of the first sentence
28 // That sentence ends after the first period followed by spa
29 // not preceded by exactly one uppercase letter. The result
30 // has no \n, \r, or \t characters and uses only single spac
31 // words.
32 //
33 func Synopsis(s string) string {
34     n := firstSentenceLen(s)
35     var b []byte
36     p := byte(' ')
37     for i := 0; i < n; i++ {
38         q := s[i]
39         if q == '\n' || q == '\r' || q == '\t' {
40             q = ' '
41         }
42         if q != ' ' || p != ' ' {
43             b = append(b, q)
44             p = q
```

```
45         }
46     }
47     // remove trailing blank, if any
48     if n := len(b); n > 0 && p == ' ' {
49         b = b[0 : n-1]
50     }
51     return string(b)
52 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/go/parser/interface.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file contains the exported entry points for invoking
6
7 package parser
8
9 import (
10     "bytes"
11     "errors"
12     "go/ast"
13     "go/token"
14     "io"
15     "io/ioutil"
16     "os"
17     "path/filepath"
18 )
19
20 // If src != nil, readSource converts src to a []byte if pos
21 // otherwise it returns an error. If src == nil, readSource
22 // the result of reading the file specified by filename.
23 //
24 func readSource(filename string, src interface{}) ([]byte, e
25     if src != nil {
26         switch s := src.(type) {
27         case string:
28             return []byte(s), nil
29         case []byte:
30             return s, nil
31         case *bytes.Buffer:
32             // is io.Reader, but src is already
33             if s != nil {
34                 return s.Bytes(), nil
35             }
36         case io.Reader:
37             var buf bytes.Buffer
38             if _, err := io.Copy(&buf, s); err !=
39                 return nil, err
40             }
41             return buf.Bytes(), nil
```

```

42         }
43         return nil, errors.New("invalid source")
44     }
45     return ioutil.ReadFile(filename)
46 }
47
48 // A Mode value is a set of flags (or 0).
49 // They control the amount of source code parsed and other o
50 // parser functionality.
51 //
52 type Mode uint
53
54 const (
55     PackageClauseOnly Mode = 1 << iota // parsing stops
56     ImportsOnly                       // parsing stops
57     ParseComments                     // parse comments
58     Trace                             // print a trace
59     DeclarationErrors                 // report declara
60     SpuriousErrors                    // report all (no
61 )
62
63 // ParseFile parses the source code of a single Go source fi
64 // the corresponding ast.File node. The source code may be p
65 // the filename of the source file, or via the src parameter
66 //
67 // If src != nil, ParseFile parses the source from src and t
68 // only used when recording position information. The type o
69 // for the src parameter must be string, []byte, or io.Reade
70 // If src == nil, ParseFile parses the file specified by fil
71 //
72 // The mode parameter controls the amount of source text par
73 // optional parser functionality. Position information is re
74 // file set fset.
75 //
76 // If the source couldn't be read, the returned AST is nil a
77 // indicates the specific failure. If the source was read bu
78 // errors were found, the result is a partial AST (with ast.
79 // representing the fragments of erroneous source code). Mul
80 // are returned via a scanner.ErrorList which is sorted by f
81 //
82 func ParseFile(fset *token.FileSet, filename string, src int
83     // get source
84     text, err := readSource(filename, src)
85     if err != nil {
86         return nil, err
87     }
88
89     // parse source
90     var p parser
91     p.init(fset, filename, text, mode)

```

```

92         f := p.parseFile()
93
94         // sort errors
95         if p.mode&SpuriousErrors == 0 {
96             p.errors.RemoveMultiples()
97         } else {
98             p.errors.Sort()
99         }
100
101         return f, p.errors.Err()
102     }
103
104     // ParseDir calls ParseFile for the files in the directory s
105     // returns a map of package name -> package AST with all the
106     // filter != nil, only the files with os.FileInfo entries pa
107     // are considered. The mode bits are passed to ParseFile unc
108     // information is recorded in the file set fset.
109     //
110     // If the directory couldn't be read, a nil map and the resp
111     // returned. If a parse error occurred, a non-nil but incomp
112     // first error encountered are returned.
113     //
114     func ParseDir(fset *token.FileSet, path string, filter func(
115         fd, err := os.Open(path)
116         if err != nil {
117             return nil, err
118         }
119         defer fd.Close()
120
121         list, err := fd.Readdir(-1)
122         if err != nil {
123             return nil, err
124         }
125
126         pkgs = make(map[string]*ast.Package)
127         for _, d := range list {
128             if filter == nil || filter(d) {
129                 filename := filepath.Join(path, d.Name)
130                 if src, err := ParseFile(fset, filename); err == nil {
131                     name := src.Name.Name
132                     pkg, found := pkgs[name]
133                     if !found {
134                         pkg = &ast.Package{
135                             Name: name,
136                             Files: make([]*ast.File, 0)
137                         }
138                         pkgs[name] = pkg
139                     }
140                     pkg.Files[filename] = src

```

```

141             } else if first == nil {
142                 first = err
143             }
144         }
145     }
146     return
147 }
148 }
149
150 // ParseExpr is a convenience function for obtaining the AST
151 // The position information recorded in the AST is undefined
152 //
153 func ParseExpr(x string) (ast.Expr, error) {
154     // parse x within the context of a complete package
155     // use //line directive for correct positions in err
156     // x alone on a separate line (handles line comments
157     // to force an error if the expression is incomplete
158     file, err := ParseFile(token.NewFileSet(), "", "pack
159     if err != nil {
160         return nil, err
161     }
162     return file.Decls[0].(*ast.FuncDecl).Body.List[0].(*
163 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/go/parser/parser.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package parser implements a parser for Go source files. It
6 // provided in a variety of forms (see the various Parse* fu
7 // output is an abstract syntax tree (AST) representing the
8 // parser is invoked through one of the Parse* functions.
9 //
10 package parser
11
12 import (
13     "fmt"
14     "go/ast"
15     "go/scanner"
16     "go/token"
17     "strconv"
18     "strings"
19     "unicode"
20 )
21
22 // The parser structure holds the parser's internal state.
23 type parser struct {
24     file      *token.File
25     errors    scanner.ErrorList
26     scanner   scanner.Scanner
27
28     // Tracing/debugging
29     mode     Mode // parsing mode
30     trace    bool // == (mode & Trace != 0)
31     indent   uint // indentation used for tracing output
32
33     // Comments
34     comments    []*ast.CommentGroup
35     leadComment *ast.CommentGroup // last lead comment
36     lineComment *ast.CommentGroup // last line comment
37
38     // Next token
39     pos token.Pos // token position
40     tok token.Token // one token look-ahead
41     lit string    // token literal
```

```

42
43     // Error recovery
44     // (used to limit the number of calls to syncXXX fun
45     // w/o making scanning progress - avoids potential e
46     // loops across multiple parser functions during err
47     syncPos token.Pos // last synchronization position
48     syncCnt int       // number of calls to syncXXX with
49
50     // Non-syntactic parser control
51     exprLev int // < 0: in control clause, >= 0: in expr
52
53     // Ordinary identifier scopes
54     pkgScope  *ast.Scope // pkgScope.Outer == ni
55     topScope  *ast.Scope // top-most scope; may
56     unresolved []*ast.Ident // unresolved identifie
57     imports   []*ast.ImportSpec // list of imports
58
59     // Label scope
60     // (maintained by open/close LabelScope)
61     labelScope *ast.Scope // label scope for curren
62     targetStack [][]*ast.Ident // stack of unresolved la
63 }
64
65 func (p *parser) init(fset *token.FileSet, filename string,
66 p.file = fset.AddFile(filename, fset.Base(), len(src
67 var m scanner.Mode
68 if mode&ParseComments != 0 {
69     m = scanner.ScanComments
70 }
71 eh := func(pos token.Position, msg string) { p.error
72 p.scanner.Init(p.file, src, eh, m)
73
74 p.mode = mode
75 p.trace = mode&Trace != 0 // for convenience (p.trac
76
77 p.next()
78
79 // set up the pkgScope here (as opposed to in parseF
80 // there are other parser entry points (ParseExpr, e
81 p.openScope()
82 p.pkgScope = p.topScope
83
84 // for the same reason, set up a label scope
85 p.openLabelScope()
86 }
87
88 // -----
89 // Scoping support
90
91 func (p *parser) openScope() {

```

```

92         p.topScope = ast.NewScope(p.topScope)
93     }
94
95     func (p *parser) closeScope() {
96         p.topScope = p.topScope.Outer
97     }
98
99     func (p *parser) openLabelScope() {
100         p.labelScope = ast.NewScope(p.labelScope)
101         p.targetStack = append(p.targetStack, nil)
102     }
103
104     func (p *parser) closeLabelScope() {
105         // resolve labels
106         n := len(p.targetStack) - 1
107         scope := p.labelScope
108         for _, ident := range p.targetStack[n] {
109             ident.Obj = scope.Lookup(ident.Name)
110             if ident.Obj == nil && p.mode&DeclarationErr
111                 p.error(ident.Pos(), fmt.Sprintf("la
112             }
113         }
114         // pop label scope
115         p.targetStack = p.targetStack[0:n]
116         p.labelScope = p.labelScope.Outer
117     }
118
119     func (p *parser) declare(decl, data interface{}, scope *ast.
120         for _, ident := range idents {
121             assert(ident.Obj == nil, "identifier already
122             obj := ast.NewObj(kind, ident.Name)
123             // remember the corresponding declaration fo
124             // errors and global variable resolution/typ
125             obj.Decl = decl
126             obj.Data = data
127             ident.Obj = obj
128             if ident.Name != "_" {
129                 if alt := scope.Insert(obj); alt !=
130                     prevDecl := ""
131                     if pos := alt.Pos(); pos.IsV
132                         prevDecl = fmt.Sprin
133                 }
134                 p.error(ident.Pos(), fmt.Spr
135             }
136         }
137     }
138 }
139
140 func (p *parser) shortVarDecl(decl *ast.AssignStmt, list []a

```

```

141 // Go spec: A short variable declaration may redecla
142 // provided they were originally declared in the sam
143 // the same type, and at least one of the non-blank
144 n := 0 // number of new variables
145 for _, x := range list {
146     if ident, isIdent := x.(*ast.Ident); isIdent
147         assert(ident.Obj == nil, "identifier
148             obj := ast.NewObj(ast.Var, ident.Nam
149             // remember corresponding assignment
150             obj.Decl = decl
151             ident.Obj = obj
152             if ident.Name != "_" {
153                 if alt := p.topScope.Insert(
154                     ident.Obj = alt // r
155                 } else {
156                     n++ // new declarati
157                 }
158             }
159         } else {
160             p.errorExpected(x.Pos(), "identifier
161         }
162     }
163     if n == 0 && p.mode&DeclarationErrors != 0 {
164         p.error(list[0].Pos(), "no new variables on
165     }
166 }
167
168 // The unresolved object is a sentinel to mark identifiers t
169 // to the list of unresolved identifiers. The sentinel is on
170 // internal consistency.
171 var unresolved = new(ast.Object)
172
173 func (p *parser) resolve(x ast.Expr) {
174     // nothing to do if x is not an identifier or the bl
175     ident, _ := x.(*ast.Ident)
176     if ident == nil {
177         return
178     }
179     assert(ident.Obj == nil, "identifier already declare
180     if ident.Name == "_" {
181         return
182     }
183     // try to resolve the identifier
184     for s := p.topScope; s != nil; s = s.Outer {
185         if obj := s.Lookup(ident.Name); obj != nil {
186             ident.Obj = obj
187             return
188         }
189     }

```



```

240             default:
241                 p.printTrace(s)
242             }
243         }
244
245         p.pos, p.tok, p.lit = p.scanner.Scan()
246     }
247
248     // Consume a comment and return it and the line on which it
249     func (p *parser) consumeComment() (comment *ast.Comment, end
250         // /*-style comments may end on a different line tha
251         // Scan the comment for '\n' chars and adjust endlin
252         endline = p.file.Line(p.pos)
253         if p.lit[1] == '*' {
254             // don't use range here - no need to decode
255             for i := 0; i < len(p.lit); i++ {
256                 if p.lit[i] == '\n' {
257                     endline++
258                 }
259             }
260         }
261
262         comment = &ast.Comment{Slash: p.pos, Text: p.lit}
263         p.next0()
264
265         return
266     }
267
268     // Consume a group of adjacent comments, add it to the parse
269     // comments list, and return it together with the line at wh
270     // the last comment in the group ends. An empty line or non-
271     // token terminates a comment group.
272     //
273     func (p *parser) consumeCommentGroup() (comments *ast.Commen
274         var list []*ast.Comment
275         endline = p.file.Line(p.pos)
276         for p.tok == token.COMMENT && endline+1 >= p.file.Li
277             var comment *ast.Comment
278             comment, endline = p.consumeComment()
279             list = append(list, comment)
280         }
281
282         // add comment group to the comments list
283         comments = &ast.CommentGroup{List: list}
284         p.comments = append(p.comments, comments)
285
286         return
287     }
288

```

```

289 // Advance to the next non-comment token. In the process, co
290 // any comment groups encountered, and remember the last lea
291 // and line comments.
292 //
293 // A lead comment is a comment group that starts and ends in
294 // line without any other tokens and that is followed by a n
295 // token on the line immediately after the comment group.
296 //
297 // A line comment is a comment group that follows a non-comm
298 // token on the same line, and that has no tokens after it o
299 // where it ends.
300 //
301 // Lead and line comments may be considered documentation th
302 // stored in the AST.
303 //
304 func (p *parser) next() {
305     p.leadComment = nil
306     p.lineComment = nil
307     line := p.file.Line(p.pos) // current line
308     p.next0()
309
310     if p.tok == token.COMMENT {
311         var comment *ast.CommentGroup
312         var endlne int
313
314         if p.file.Line(p.pos) == line {
315             // The comment is on same line as th
316             // cannot be a lead comment but may
317             comment, endlne = p.consumeCommentG
318             if p.file.Line(p.pos) != endlne {
319                 // The next token is on a di
320                 // the last comment group is
321                 p.lineComment = comment
322             }
323         }
324
325         // consume successor comments, if any
326         endlne = -1
327         for p.tok == token.COMMENT {
328             comment, endlne = p.consumeCommentG
329         }
330
331         if endlne+1 == p.file.Line(p.pos) {
332             // The next token is following on th
333             // comment group, thus the last comm
334             p.leadComment = comment
335         }
336     }
337 }

```

```

338
339 func (p *parser) error(pos token.Pos, msg string) {
340     p.errors.Add(p.file.Position(pos), msg)
341 }
342
343 func (p *parser) errorExpected(pos token.Pos, msg string) {
344     msg = "expected " + msg
345     if pos == p.pos {
346         // the error happened at the current position
347         // make the error message more specific
348         if p.tok == token.SEMICOLON && p.lit == "\n"
349             msg += ", found newline"
350     } else {
351         msg += ", found '" + p.tok.String()
352         if p.tok.IsLiteral() {
353             msg += " " + p.lit
354         }
355     }
356 }
357 p.error(pos, msg)
358 }
359
360 func (p *parser) expect(tok token.Token) token.Pos {
361     pos := p.pos
362     if p.tok != tok {
363         p.errorExpected(pos, ""+tok.String()+"")
364     }
365     p.next() // make progress
366     return pos
367 }
368
369 // expectClosing is like expect but provides a better error
370 // for the common case of a missing comma before a newline.
371 //
372 func (p *parser) expectClosing(tok token.Token, context string) token.Pos {
373     if p.tok != tok && p.tok == token.SEMICOLON && p.lit != "" {
374         p.error(p.pos, "missing ',' before newline in "+context)
375         p.next()
376     }
377     return p.expect(tok)
378 }
379
380 func (p *parser) expectSemi() {
381     // semicolon is optional before a closing ')' or '}'
382     if p.tok != token.RPAREN && p.tok != token.RBRACE {
383         if p.tok == token.SEMICOLON {
384             p.next()
385         } else {
386             p.errorExpected(p.pos, "';'")
387             syncStmt(p)
388         }
389     }
390 }

```

```

388         }
389     }
390 }
391
392 func (p *parser) atComma(context string) bool {
393     if p.tok == token.COMMA {
394         return true
395     }
396     if p.tok == token.SEMICOLON && p.lit == "\n" {
397         p.error(p.pos, "missing ',' before newline i
398         return true // "insert" the comma and contin
399
400     }
401     return false
402 }
403
404 func assert(cond bool, msg string) {
405     if !cond {
406         panic("go/parser internal error: " + msg)
407     }
408 }
409
410 // syncStmt advances to the next statement.
411 // Used for synchronization after an error.
412 //
413 func syncStmt(p *parser) {
414     for {
415         switch p.tok {
416             case token.BREAK, token.CONST, token.CONTINU
417                 token.FALLTHROUGH, token.FOR, token.
418                 token.IF, token.RETURN, token.SELECT
419                 token.TYPE, token.VAR:
420                 // Return only if parser made some p
421                 // sync or if it has not reached 10
422                 // progress. Otherwise consume at le
423                 // avoid an endless parser loop (it
424                 // both parseOperand and parseStmt c
425                 // correctly do not advance, thus th
426                 // invocation limit p.syncCnt).
427                 if p.pos == p.syncPos && p.syncCnt <
428                     p.syncCnt++
429                     return
430             }
431             if p.pos > p.syncPos {
432                 p.syncPos = p.pos
433                 p.syncCnt = 0
434                 return
435             }
436             // Reaching here indicates a parser

```

```

437             // incorrect token list in this func
438             // leads to skipping of possibly cor
439             // previous error is present, and th
440             // over a non-terminating parse.
441             case token.EOF:
442                 return
443             }
444             p.next()
445         }
446     }
447
448     // syncDecl advances to the next declaration.
449     // Used for synchronization after an error.
450     //
451     func syncDecl(p *parser) {
452         for {
453             switch p.tok {
454             case token.CONST, token.TYPE, token.VAR:
455                 // see comments in syncStmt
456                 if p.pos == p.syncPos && p.syncCnt <
457                     p.syncCnt++
458                     return
459             }
460             if p.pos > p.syncPos {
461                 p.syncPos = p.pos
462                 p.syncCnt = 0
463                 return
464             }
465             case token.EOF:
466                 return
467             }
468             p.next()
469         }
470     }
471
472     // -----
473     // Identifiers
474
475     func (p *parser) parseIdent() *ast.Ident {
476         pos := p.pos
477         name := "_"
478         if p.tok == token.IDENT {
479             name = p.lit
480             p.next()
481         } else {
482             p.expect(token.IDENT) // use expect() error
483         }
484         return &ast.Ident{NamePos: pos, Name: name}
485     }

```

```

486
487 func (p *parser) parseIdentList() (list []*ast.Ident) {
488     if p.trace {
489         defer un(trace(p, "IdentList"))
490     }
491
492     list = append(list, p.parseIdent())
493     for p.tok == token.COMMA {
494         p.next()
495         list = append(list, p.parseIdent())
496     }
497
498     return
499 }
500
501 // -----
502 // Common productions
503
504 // If lhs is set, result list elements which are identifiers
505 func (p *parser) parseExprList(lhs bool) (list []*ast.Expr) {
506     if p.trace {
507         defer un(trace(p, "ExpressionList"))
508     }
509
510     list = append(list, p.checkExpr(p.parseExpr(lhs)))
511     for p.tok == token.COMMA {
512         p.next()
513         list = append(list, p.checkExpr(p.parseExpr(
514     }
515
516     return
517 }
518
519 func (p *parser) parseLhsList() []*ast.Expr {
520     list := p.parseExprList(true)
521     switch p.tok {
522     case token.DEFINE:
523         // lhs of a short variable declaration
524         // but doesn't enter scope until later:
525         // caller must call p.shortVarDecl(p.makeIde
526         // at appropriate time.
527     case token.COLON:
528         // lhs of a label declaration or a communica
529         // statement (parseLhsList is not called whe
530         // of a switch statement):
531         // - labels are declared by the caller of pa
532         // - for communication clauses, if there is
533         // followed by a colon, we have a syntax e
534         // to resolve the identifier in that case
535     default:

```

```

536             // identifiers must be declared elsewhere
537             for _, x := range list {
538                 p.resolve(x)
539             }
540         }
541         return list
542     }
543
544     func (p *parser) parseRhsList() []ast.Expr {
545         return p.parseExprList(false)
546     }
547
548     // -----
549     // Types
550
551     func (p *parser) parseType() ast.Expr {
552         if p.trace {
553             defer un(trace(p, "Type"))
554         }
555
556         typ := p.tryType()
557
558         if typ == nil {
559             pos := p.pos
560             p.errorExpected(pos, "type")
561             p.next() // make progress
562             return &ast.BadExpr{From: pos, To: p.pos}
563         }
564
565         return typ
566     }
567
568     // If the result is an identifier, it is not resolved.
569     func (p *parser) parseTypeName() ast.Expr {
570         if p.trace {
571             defer un(trace(p, "TypeName"))
572         }
573
574         ident := p.parseIdent()
575         // don't resolve ident yet - it may be a parameter o
576
577         if p.tok == token.PERIOD {
578             // ident is a package name
579             p.next()
580             p.resolve(ident)
581             sel := p.parseIdent()
582             return &ast.SelectorExpr{X: ident, Sel: sel}
583         }
584

```

```

585         return ident
586     }
587
588     func (p *parser) parseArrayType(ellipsisOk bool) ast.Expr {
589         if p.trace {
590             defer un(trace(p, "ArrayType"))
591         }
592
593         lbrack := p.expect(token.LBRACK)
594         var len ast.Expr
595         if ellipsisOk && p.tok == token.ELLIPSIS {
596             len = &ast.Ellipsis{Ellipsis: p.pos}
597             p.next()
598         } else if p.tok != token.RBRACK {
599             len = p.parseRhs()
600         }
601         p.expect(token.RBRACK)
602         elt := p.parseType()
603
604         return &ast.ArrayType{Lbrack: lbrack, Len: len, Elt:
605     }
606
607     func (p *parser) makeIdentList(list []ast.Expr) []*ast.Ident
608         idents := make([]*ast.Ident, len(list))
609         for i, x := range list {
610             ident, isIdent := x.(*ast.Ident)
611             if !isIdent {
612                 if _, isBad := x.(*ast.BadExpr); !is
613                     // only report error if it's
614                     p.errorExpected(x.Pos(), "id
615                 }
616                 ident = &ast.Ident{NamePos: x.Pos(),
617             }
618             idents[i] = ident
619         }
620         return idents
621     }
622
623     func (p *parser) parseFieldDecl(scope *ast.Scope) *ast.Field
624         if p.trace {
625             defer un(trace(p, "FieldDecl"))
626         }
627
628         doc := p.leadComment
629
630         // fields
631         list, typ := p.parseVarList(false)
632
633         // optional tag

```

```

634     var tag *ast.BasicLit
635     if p.tok == token.STRING {
636         tag = &ast.BasicLit{ValuePos: p.pos, Kind: p
637     }
638     }
639
640     // analyze case
641     var idents []*ast.Ident
642     if typ != nil {
643         // IdentifierList Type
644         idents = p.makeIdentList(list)
645     } else {
646         // ["*"] TypeName (AnonymousField)
647         typ = list[0] // we always have at least one
648         p.resolve(typ)
649         if n := len(list); n > 1 || !isTypeName(dere
650             pos := typ.Pos()
651             p.errorExpected(pos, "anonymous fiel
652             typ = &ast.BadExpr{From: pos, To: li
653         }
654     }
655
656     p.expectSemi() // call before accessing p.linecommen
657
658     field := &ast.Field{Doc: doc, Names: idents, Type: t
659     p.declare(field, nil, scope, ast.Var, idents...)
660
661     return field
662 }
663
664 func (p *parser) parseStructType() *ast.StructType {
665     if p.trace {
666         defer un(trace(p, "StructType"))
667     }
668
669     pos := p.expect(token.STRUCT)
670     lbrace := p.expect(token.LBRACE)
671     scope := ast.NewScope(nil) // struct scope
672     var list []*ast.Field
673     for p.tok == token.IDENT || p.tok == token.MUL || p.
674         // a field declaration cannot start with a '
675         // it here for more robust parsing and bette
676         // (parseFieldDecl will check and complain i
677         list = append(list, p.parseFieldDecl(scope))
678     }
679     rbrace := p.expect(token.RBRACE)
680
681     return &ast.StructType{
682         Struct: pos,
683         Fields: &ast.FieldList{

```

```

684             Opening: lbrace,
685             List:    list,
686             Closing: rbrace,
687         },
688     }
689 }
690
691 func (p *parser) parsePointerType() *ast.StarExpr {
692     if p.trace {
693         defer un(trace(p, "PointerType"))
694     }
695
696     star := p.expect(token.MUL)
697     base := p.parseType()
698
699     return &ast.StarExpr{Star: star, X: base}
700 }
701
702 func (p *parser) tryVarType(isParam bool) ast.Expr {
703     if isParam && p.tok == token.ELLIPSIS {
704         pos := p.pos
705         p.next()
706         typ := p.tryIdentOrType(isParam) // don't us
707         if typ == nil {
708             p.error(pos, "'...' parameter is mis
709                 typ = &ast.BadExpr{From: pos, To: p.
710         }
711         return &ast.Ellipsis{Ellipsis: pos, Elt: typ
712     }
713     return p.tryIdentOrType(false)
714 }
715
716 func (p *parser) parseVarType(isParam bool) ast.Expr {
717     typ := p.tryVarType(isParam)
718     if typ == nil {
719         pos := p.pos
720         p.errorExpected(pos, "type")
721         p.next() // make progress
722         typ = &ast.BadExpr{From: pos, To: p.pos}
723     }
724     return typ
725 }
726
727 func (p *parser) parseVarList(isParam bool) (list []ast.Expr
728     if p.trace {
729         defer un(trace(p, "VarList"))
730     }
731
732     // a list of identifiers looks like a list of type n

```

```

733         //
734         // parse/tryVarType accepts any type (including pare
735         // ones) even though the syntax does not permit them
736         // accept them all for more robust parsing and compl
737         for typ := p.parseVarType(isParam); typ != nil; {
738             list = append(list, typ)
739             if p.tok != token.COMMA {
740                 break
741             }
742             p.next()
743             typ = p.tryVarType(isParam) // maybe nil as
744         }
745
746         // if we had a list of identifiers, it must be follo
747         if typ = p.tryVarType(isParam); typ != nil {
748             p.resolve(typ)
749         }
750
751         return
752     }
753
754     func (p *parser) parseParameterList(scope *ast.Scope, ellips
755         if p.trace {
756             defer un(trace(p, "ParameterList"))
757         }
758
759         list, typ := p.parseVarList(ellipsisOk)
760         if typ != nil {
761             // IdentifierList Type
762             idents := p.makeIdentList(list)
763             field := &ast.Field{Names: idents, Type: typ
764             params = append(params, field)
765             // Go spec: The scope of an identifier denot
766             // parameter or result variable is the funct
767             p.declare(field, nil, scope, ast.Var, idents
768             if p.tok == token.COMMA {
769                 p.next()
770             }
771
772             for p.tok != token.RPAREN && p.tok != token.
773                 idents := p.parseIdentList()
774                 typ := p.parseVarType(ellipsisOk)
775                 field := &ast.Field{Names: idents, T
776                 params = append(params, field)
777                 // Go spec: The scope of an identifi
778                 // parameter or result variable is t
779                 p.declare(field, nil, scope, ast.Var
780                 if !p.atComma("parameter list") {
781                     break

```

```

782         }
783         p.next()
784     }
785
786     } else {
787         // Type { ", " Type } (anonymous parameters)
788         params = make([]*ast.Field, len(list))
789         for i, x := range list {
790             p.resolve(x)
791             params[i] = &ast.Field{Type: x}
792         }
793     }
794
795     return
796 }
797
798 func (p *parser) parseParameters(scope *ast.Scope, ellipsisC
799     if p.trace {
800         defer un(trace(p, "Parameters"))
801     }
802
803     var params []*ast.Field
804     lparen := p.expect(token.LPAREN)
805     if p.tok != token.RPAREN {
806         params = p.parseParameterList(scope, ellipsi
807     }
808     rparen := p.expect(token.RPAREN)
809
810     return &ast.FieldList{Opening: lparen, List: params,
811 }
812
813 func (p *parser) parseResult(scope *ast.Scope) *ast.FieldLis
814     if p.trace {
815         defer un(trace(p, "Result"))
816     }
817
818     if p.tok == token.LPAREN {
819         return p.parseParameters(scope, false)
820     }
821
822     typ := p.tryType()
823     if typ != nil {
824         list := make([]*ast.Field, 1)
825         list[0] = &ast.Field{Type: typ}
826         return &ast.FieldList{List: list}
827     }
828
829     return nil
830 }
831

```

```

832 func (p *parser) parseSignature(scope *ast.Scope) (params, r
833     if p.trace {
834         defer un(trace(p, "Signature"))
835     }
836
837     params = p.parseParameters(scope, true)
838     results = p.parseResult(scope)
839
840     return
841 }
842
843 func (p *parser) parseFuncType() (*ast.FuncType, *ast.Scope)
844     if p.trace {
845         defer un(trace(p, "FuncType"))
846     }
847
848     pos := p.expect(token.FUNC)
849     scope := ast.NewScope(p.topScope) // function scope
850     params, results := p.parseSignature(scope)
851
852     return &ast.FuncType{Func: pos, Params: params, Resu
853 }
854
855 func (p *parser) parseMethodSpec(scope *ast.Scope) *ast.Fiel
856     if p.trace {
857         defer un(trace(p, "MethodSpec"))
858     }
859
860     doc := p.leadComment
861     var idents []*ast.Ident
862     var typ ast.Expr
863     x := p.parseTypeName()
864     if ident, isIdent := x.(*ast.Ident); isIdent && p.to
865         // method
866         idents = []*ast.Ident{ident}
867         scope := ast.NewScope(nil) // method scope
868         params, results := p.parseSignature(scope)
869         typ = &ast.FuncType{Func: token.NoPos, Param
870     } else {
871         // embedded interface
872         typ = x
873         p.resolve(typ)
874     }
875     p.expectSemi() // call before accessing p.linecommen
876
877     spec := &ast.Field{Doc: doc, Names: idents, Type: ty
878     p.declare(spec, nil, scope, ast.Fun, idents...)
879
880     return spec

```

```

881 }
882
883 func (p *parser) parseInterfaceType() *ast.InterfaceType {
884     if p.trace {
885         defer un(trace(p, "InterfaceType"))
886     }
887
888     pos := p.expect(token.INTERFACE)
889     lbrace := p.expect(token.LBRACE)
890     scope := ast.NewScope(nil) // interface scope
891     var list []*ast.Field
892     for p.tok == token.IDENT {
893         list = append(list, p.parseMethodSpec(scope))
894     }
895     rbrace := p.expect(token.RBRACE)
896
897     return &ast.InterfaceType{
898         Interface: pos,
899         Methods: &ast.FieldList{
900             Opening: lbrace,
901             List:    list,
902             Closing: rbrace,
903         },
904     }
905 }
906
907 func (p *parser) parseMapType() *ast.MapType {
908     if p.trace {
909         defer un(trace(p, "MapType"))
910     }
911
912     pos := p.expect(token.MAP)
913     p.expect(token.LBRACK)
914     key := p.parseType()
915     p.expect(token.RBRACK)
916     value := p.parseType()
917
918     return &ast.MapType{Map: pos, Key: key, Value: value}
919 }
920
921 func (p *parser) parseChanType() *ast.ChanType {
922     if p.trace {
923         defer un(trace(p, "ChanType"))
924     }
925
926     pos := p.pos
927     dir := ast.SEND | ast.RECV
928     if p.tok == token.CHAN {
929         p.next()

```

```

930         if p.tok == token.ARROW {
931             p.next()
932             dir = ast.SEND
933         }
934     } else {
935         p.expect(token.ARROW)
936         p.expect(token.CHAN)
937         dir = ast.RECV
938     }
939     value := p.parseType()
940
941     return &ast.ChanType{Begin: pos, Dir: dir, Value: va
942 }
943
944 // If the result is an identifier, it is not resolved.
945 func (p *parser) tryIdentOrType(ellipsisOk bool) ast.Expr {
946     switch p.tok {
947     case token.IDENT:
948         return p.parseTypeName()
949     case token.LBRACK:
950         return p.parseArrayType(ellipsisOk)
951     case token.STRUCT:
952         return p.parseStructType()
953     case token.MUL:
954         return p.parsePointerType()
955     case token.FUNC:
956         typ, _ := p.parseFuncType()
957         return typ
958     case token.INTERFACE:
959         return p.parseInterfaceType()
960     case token.MAP:
961         return p.parseMapType()
962     case token.CHAN, token.ARROW:
963         return p.parseChanType()
964     case token.LPAREN:
965         lparen := p.pos
966         p.next()
967         typ := p.parseType()
968         rparen := p.expect(token.RPAREN)
969         return &ast.ParenExpr{Lparen: lparen, X: typ
970     }
971
972     // no type found
973     return nil
974 }
975
976 func (p *parser) tryType() ast.Expr {
977     typ := p.tryIdentOrType(false)
978     if typ != nil {
979         p.resolve(typ)

```

```

980         }
981         return typ
982     }
983
984     // -----
985     // Blocks
986
987     func (p *parser) parseStmtList() (list []ast.Stmt) {
988         if p.trace {
989             defer un(trace(p, "StatementList"))
990         }
991
992         for p.tok != token.CASE && p.tok != token.DEFAULT &&
993             list = append(list, p.parseStmt())
994     }
995
996     return
997 }
998
999 func (p *parser) parseBody(scope *ast.Scope) *ast.BlockStmt
1000     if p.trace {
1001         defer un(trace(p, "Body"))
1002     }
1003
1004     lbrace := p.expect(token.LBRACE)
1005     p.topScope = scope // open function scope
1006     p.openLabelScope()
1007     list := p.parseStmtList()
1008     p.closeLabelScope()
1009     p.closeScope()
1010     rbrace := p.expect(token.RBRACE)
1011
1012     return &ast.BlockStmt{Lbrace: lbrace, List: list, Rb
1013 }
1014
1015 func (p *parser) parseBlockStmt() *ast.BlockStmt {
1016     if p.trace {
1017         defer un(trace(p, "BlockStmt"))
1018     }
1019
1020     lbrace := p.expect(token.LBRACE)
1021     p.openScope()
1022     list := p.parseStmtList()
1023     p.closeScope()
1024     rbrace := p.expect(token.RBRACE)
1025
1026     return &ast.BlockStmt{Lbrace: lbrace, List: list, Rb
1027 }
1028

```

```

1029 // -----
1030 // Expressions
1031
1032 func (p *parser) parseFuncTypeOrLit() ast.Expr {
1033     if p.trace {
1034         defer un(trace(p, "FuncTypeOrLit"))
1035     }
1036
1037     typ, scope := p.parseFuncType()
1038     if p.tok != token.LBRACE {
1039         // function type only
1040         return typ
1041     }
1042
1043     p.exprLev++
1044     body := p.parseBody(scope)
1045     p.exprLev--
1046
1047     return &ast.FuncLit{Type: typ, Body: body}
1048 }
1049
1050 // parseOperand may return an expression or a raw type (incl
1051 // types of the form [...]T. Callers must verify the result.
1052 // If lhs is set and the result is an identifier, it is not
1053 //
1054 func (p *parser) parseOperand(lhs bool) ast.Expr {
1055     if p.trace {
1056         defer un(trace(p, "Operand"))
1057     }
1058
1059     switch p.tok {
1060     case token.IDENT:
1061         x := p.parseIdent()
1062         if !lhs {
1063             p.resolve(x)
1064         }
1065         return x
1066
1067     case token.INT, token.FLOAT, token.IMAG, token.CHAR,
1068         x := &ast.BasicLit{ValuePos: p.pos, Kind: p.
1069         p.next()
1070         return x
1071
1072     case token.LPAREN:
1073         lparen := p.pos
1074         p.next()
1075         p.exprLev++
1076         x := p.parseRhsOrType() // types may be pare
1077         p.exprLev--

```

```

1078         rparen := p.expect(token.RPAREN)
1079         return &ast.ParenExpr{Lparen: lparen, X: x,
1080
1081     case token.FUNC:
1082         return p.parseFuncTypeOrLit()
1083     }
1084
1085     if typ := p.tryIdentOrType(true); typ != nil {
1086         // could be type for composite literal or co
1087         _, isIdent := typ.(*ast.Ident)
1088         assert(!isIdent, "type cannot be identifier")
1089         return typ
1090     }
1091
1092     // we have an error
1093     pos := p.pos
1094     p.errorExpected(pos, "operand")
1095     syncStmt(p)
1096     return &ast.BadExpr{From: pos, To: p.pos}
1097 }
1098
1099 func (p *parser) parseSelector(x ast.Expr) ast.Expr {
1100     if p.trace {
1101         defer un(trace(p, "Selector"))
1102     }
1103
1104     sel := p.parseIdent()
1105
1106     return &ast.SelectorExpr{X: x, Sel: sel}
1107 }
1108
1109 func (p *parser) parseTypeAssertion(x ast.Expr) ast.Expr {
1110     if p.trace {
1111         defer un(trace(p, "TypeAssertion"))
1112     }
1113
1114     p.expect(token.LPAREN)
1115     var typ ast.Expr
1116     if p.tok == token.TYPE {
1117         // type switch: typ == nil
1118         p.next()
1119     } else {
1120         typ = p.parseType()
1121     }
1122     p.expect(token.RPAREN)
1123
1124     return &ast.TypeAssertExpr{X: x, Type: typ}
1125 }
1126
1127 func (p *parser) parseIndexOrSlice(x ast.Expr) ast.Expr {

```

```

1128     if p.trace {
1129         defer un(trace(p, "IndexOrSlice"))
1130     }
1131
1132     lbrack := p.expect(token.LBRACK)
1133     p.exprLev++
1134     var low, high ast.Expr
1135     isSlice := false
1136     if p.tok != token.COLON {
1137         low = p.parseRhs()
1138     }
1139     if p.tok == token.COLON {
1140         isSlice = true
1141         p.next()
1142         if p.tok != token.RBRACK {
1143             high = p.parseRhs()
1144         }
1145     }
1146     p.exprLev--
1147     rbrack := p.expect(token.RBRACK)
1148
1149     if isSlice {
1150         return &ast.SliceExpr{X: x, Lbrack: lbrack,
1151     }
1152     return &ast.IndexExpr{X: x, Lbrack: lbrack, Index: l
1153 }
1154
1155 func (p *parser) parseCallOrConversion(fun ast.Expr) *ast.Call
1156 if p.trace {
1157     defer un(trace(p, "CallOrConversion"))
1158 }
1159
1160 lparen := p.expect(token.LPAREN)
1161 p.exprLev++
1162 var list []ast.Expr
1163 var ellipsis token.Pos
1164 for p.tok != token.RPAREN && p.tok != token.EOF && !
1165     list = append(list, p.parseRhsOrType()) // b
1166     if p.tok == token.ELLIPSIS {
1167         ellipsis = p.pos
1168         p.next()
1169     }
1170     if !p.atComma("argument list") {
1171         break
1172     }
1173     p.next()
1174 }
1175 p.exprLev--
1176 rparen := p.expectClosing(token.RPAREN, "argument li

```

```

1177
1178     return &ast.CallExpr{Fun: fun, Lparen: lparen, Args:
1179 }
1180
1181 func (p *parser) parseElement(keyOk bool) ast.Expr {
1182     if p.trace {
1183         defer un(trace(p, "Element"))
1184     }
1185
1186     if p.tok == token.LBRACE {
1187         return p.parseLiteralValue(nil)
1188     }
1189
1190     x := p.checkExpr(p.parseExpr(keyOk)) // don't resolv
1191     if keyOk {
1192         if p.tok == token.COLON {
1193             colon := p.pos
1194             p.next()
1195             return &ast.KeyValueExpr{Key: x, Col
1196         }
1197         p.resolve(x) // not a map key
1198     }
1199
1200     return x
1201 }
1202
1203 func (p *parser) parseElementList() (list []ast.Expr) {
1204     if p.trace {
1205         defer un(trace(p, "ElementList"))
1206     }
1207
1208     for p.tok != token.RBRACE && p.tok != token.EOF {
1209         list = append(list, p.parseElement(true))
1210         if !p.atComma("composite literal") {
1211             break
1212         }
1213         p.next()
1214     }
1215
1216     return
1217 }
1218
1219 func (p *parser) parseLiteralValue(typ ast.Expr) ast.Expr {
1220     if p.trace {
1221         defer un(trace(p, "LiteralValue"))
1222     }
1223
1224     lbrace := p.expect(token.LBRACE)
1225     var elts []ast.Expr

```

```

1226         p.exprLev++
1227         if p.tok != token.RBRACE {
1228             elts = p.parseElementList()
1229         }
1230         p.exprLev--
1231         rbrace := p.expectClosing(token.RBRACE, "composite l
1232         return &ast.CompositeLit{Type: typ, Lbrace: lbrace,
1233     }
1234
1235     // checkExpr checks that x is an expression (and not a type)
1236     func (p *parser) checkExpr(x ast.Expr) ast.Expr {
1237         switch unparen(x).(type) {
1238             case *ast.BadExpr:
1239             case *ast.Ident:
1240             case *ast.BasicLit:
1241             case *ast.FuncLit:
1242             case *ast.CompositeLit:
1243             case *ast.ParenExpr:
1244                 panic("unreachable")
1245             case *ast.SelectorExpr:
1246             case *ast.IndexExpr:
1247             case *ast.SliceExpr:
1248             case *ast.TypeAssertExpr:
1249                 // If t.Type == nil we have a type assertion
1250                 // y.(type), which is only allowed in type s
1251                 // It's hard to exclude those but for the ca
1252                 // a type switch. Instead be lenient and tes
1253                 // checker.
1254             case *ast.CallExpr:
1255             case *ast.StarExpr:
1256             case *ast.UnaryExpr:
1257             case *ast.BinaryExpr:
1258             default:
1259                 // all other nodes are not proper expression
1260                 p.errorExpected(x.Pos(), "expression")
1261                 x = &ast.BadExpr{From: x.Pos(), To: x.End()}
1262         }
1263         return x
1264     }
1265
1266     // isTypeName returns true iff x is a (qualified) TypeName.
1267     func isTypeName(x ast.Expr) bool {
1268         switch t := x.(type) {
1269             case *ast.BadExpr:
1270             case *ast.Ident:
1271             case *ast.SelectorExpr:
1272                 _, isIdent := t.X.(*ast.Ident)
1273                 return isIdent
1274             default:
1275                 return false // all other nodes are not type

```

```

1276     }
1277     return true
1278 }
1279
1280 // isLiteralType returns true iff x is a legal composite lit
1281 func isLiteralType(x ast.Expr) bool {
1282     switch t := x.(type) {
1283     case *ast.BadExpr:
1284     case *ast.Ident:
1285     case *ast.SelectorExpr:
1286         _, isIdent := t.X.(*ast.Ident)
1287         return isIdent
1288     case *ast.ArrayType:
1289     case *ast.StructType:
1290     case *ast.MapType:
1291     default:
1292         return false // all other nodes are not legal
1293     }
1294     return true
1295 }
1296
1297 // If x is of the form *T, deref returns T, otherwise it returns x
1298 func deref(x ast.Expr) ast.Expr {
1299     if p, isPtr := x.(*ast.StarExpr); isPtr {
1300         x = p.X
1301     }
1302     return x
1303 }
1304
1305 // If x is of the form (T), unparen returns unparen(T), otherwise it returns x
1306 func unparen(x ast.Expr) ast.Expr {
1307     if p, isParen := x.(*ast.ParenExpr); isParen {
1308         x = unparen(p.X)
1309     }
1310     return x
1311 }
1312
1313 // checkExprOrType checks that x is an expression or a type
1314 // (and not a raw type such as [...]T).
1315 //
1316 func (p *parser) checkExprOrType(x ast.Expr) ast.Expr {
1317     switch t := unparen(x).(type) {
1318     case *ast.ParenExpr:
1319         panic("unreachable")
1320     case *ast.UnaryExpr:
1321     case *ast.ArrayType:
1322         if len, isEllipsis := t.Len.(*ast.Ellipsis);
1323         p.error(len.Pos(), "expected array length") {
1324             x = &ast.BadExpr{From: x.Pos(), To:

```

```

1325         }
1326     }
1327
1328     // all other nodes are expressions or types
1329     return x
1330 }
1331
1332 // If lhs is set and the result is an identifier, it is not
1333 func (p *parser) parsePrimaryExpr(lhs bool) ast.Expr {
1334     if p.trace {
1335         defer un(trace(p, "PrimaryExpr"))
1336     }
1337
1338     x := p.parseOperand(lhs)
1339 L:
1340     for {
1341         switch p.tok {
1342         case token.PERIOD:
1343             p.next()
1344             if lhs {
1345                 p.resolve(x)
1346             }
1347             switch p.tok {
1348             case token.IDENT:
1349                 x = p.parseSelector(p.checkE
1350             case token.LPAREN:
1351                 x = p.parseTypeAssertion(p.c
1352             default:
1353                 pos := p.pos
1354                 p.errorExpected(pos, "select
1355                 p.next() // make progress
1356                 x = &ast.BadExpr{From: pos,
1357             }
1358         case token.LBRACK:
1359             if lhs {
1360                 p.resolve(x)
1361             }
1362             x = p.parseIndexOrSlice(p.checkExpr(
1363         case token.LPAREN:
1364             if lhs {
1365                 p.resolve(x)
1366             }
1367             x = p.parseCallOrConversion(p.checkE
1368         case token.LBRACE:
1369             if isLiteralType(x) && (p.exprLev >=
1370                 if lhs {
1371                     p.resolve(x)
1372                 }
1373             x = p.parseLiteralValue(x)

```

```

1374             } else {
1375                 break L
1376             }
1377         default:
1378             break L
1379     }
1380     lhs = false // no need to try to resolve aga
1381 }
1382
1383     return x
1384 }
1385
1386 // If lhs is set and the result is an identifier, it is not
1387 func (p *parser) parseUnaryExpr(lhs bool) ast.Expr {
1388     if p.trace {
1389         defer un(trace(p, "UnaryExpr"))
1390     }
1391
1392     switch p.tok {
1393     case token.ADD, token.SUB, token.NOT, token.XOR, tok
1394         pos, op := p.pos, p.tok
1395         p.next()
1396         x := p.parseUnaryExpr(false)
1397         return &ast.UnaryExpr{OpPos: pos, Op: op, X:
1398
1399     case token.ARROW:
1400         // channel type or receive expression
1401         pos := p.pos
1402         p.next()
1403         if p.tok == token.CHAN {
1404             p.next()
1405             value := p.parseType()
1406             return &ast.ChanType{Begin: pos, Dir
1407         }
1408
1409         x := p.parseUnaryExpr(false)
1410         return &ast.UnaryExpr{OpPos: pos, Op: token.
1411
1412     case token.MUL:
1413         // pointer type or unary "*" expression
1414         pos := p.pos
1415         p.next()
1416         x := p.parseUnaryExpr(false)
1417         return &ast.StarExpr{Star: pos, X: p.checkEx
1418     }
1419
1420     return p.parsePrimaryExpr(lhs)
1421 }
1422
1423 // If lhs is set and the result is an identifier, it is not

```

```

1424 func (p *parser) parseBinaryExpr(lhs bool, prec1 int) ast.Ex
1425     if p.trace {
1426         defer un(trace(p, "BinaryExpr"))
1427     }
1428
1429     x := p.parseUnaryExpr(lhs)
1430     for prec := p.tok.Precedence(); prec >= prec1; prec-
1431         for p.tok.Precedence() == prec {
1432             pos, op := p.pos, p.tok
1433             p.next()
1434             if lhs {
1435                 p.resolve(x)
1436                 lhs = false
1437             }
1438             y := p.parseBinaryExpr(false, prec+1
1439             x = &ast.BinaryExpr{X: p.checkExpr(x
1440         }
1441     }
1442
1443     return x
1444 }
1445
1446 // If lhs is set and the result is an identifier, it is not
1447 // The result may be a type or even a raw type ([...]int). C
1448 // check the result (using checkExpr or checkExprOrType), de
1449 // context.
1450 func (p *parser) parseExpr(lhs bool) ast.Expr {
1451     if p.trace {
1452         defer un(trace(p, "Expression"))
1453     }
1454
1455     return p.parseBinaryExpr(lhs, token.LowestPrec+1)
1456 }
1457
1458 func (p *parser) parseRhs() ast.Expr {
1459     return p.checkExpr(p.parseExpr(false))
1460 }
1461
1462 func (p *parser) parseRhsOrType() ast.Expr {
1463     return p.checkExprOrType(p.parseExpr(false))
1464 }
1465
1466 // -----
1467 // Statements
1468
1469 // Parsing modes for parseSimpleStmt.
1470 const (
1471     basic = iota
1472     labelOk

```

```

1473         rangeOk
1474     )
1475
1476 // parseSimpleStmt returns true as 2nd result if it parsed t
1477 // of a range clause (with mode == rangeOk). The returned st
1478 // assignment with a right-hand side that is a single unary
1479 // the form "range x". No guarantees are given for the left-
1480 func (p *parser) parseSimpleStmt(mode int) (ast.Stmt, bool)
1481     if p.trace {
1482         defer un(trace(p, "SimpleStmt"))
1483     }
1484
1485     x := p.parseLhsList()
1486
1487     switch p.tok {
1488     case
1489         token.DEFINE, token.ASSIGN, token.ADD_ASSIGN
1490         token.SUB_ASSIGN, token.MUL_ASSIGN, token.QU
1491         token.REM_ASSIGN, token.AND_ASSIGN, token.OR
1492         token.XOR_ASSIGN, token.SHL_ASSIGN, token.SH
1493         // assignment statement, possibly part of a
1494         pos, tok := p.pos, p.tok
1495         p.next()
1496         var y []ast.Expr
1497         isRange := false
1498         if mode == rangeOk && p.tok == token.RANGE &
1499             pos := p.pos
1500             p.next()
1501             y = []ast.Expr{&ast.UnaryExpr{OpPos:
1502                 isRange = true
1503             } else {
1504                 y = p.parseRhsList()
1505             }
1506             as := &ast.AssignStmt{Lhs: x, TokPos: pos, T
1507             if tok == token.DEFINE {
1508                 p.shortVarDecl(as, x)
1509             }
1510             return as, isRange
1511     }
1512
1513     if len(x) > 1 {
1514         p.errorExpected(x[0].Pos(), "1 expression")
1515         // continue with first expression
1516     }
1517
1518     switch p.tok {
1519     case token.COLON:
1520         // labeled statement
1521         colon := p.pos

```

```

1522         p.next()
1523         if label, isIdent := x[0].(*ast.Ident); mode
1524             // Go spec: The scope of a label is
1525             // in which it is declared and exclu
1526             // function.
1527             stmt := &ast.LabeledStmt{Label: labe
1528             p.declare(stmt, nil, p.labelScope, a
1529             return stmt, false
1530         }
1531         // The label declaration typically starts at
1532         // declaration may be erroneous due to a tok
1533         // before the ':'). If SpuriousErrors is not
1534         // ported for the line is the illegal label
1535         // before the ':' that caused the problem. T
1536         // position for error reporting.
1537         p.error(colon, "illegal label declaration")
1538         return &ast.BadStmt{From: x[0].Pos(), To: co
1539
1540     case token.ARROW:
1541         // send statement
1542         arrow := p.pos
1543         p.next()
1544         y := p.parseRhs()
1545         return &ast.SendStmt{Chan: x[0], Arrow: arro
1546
1547     case token.INC, token.DEC:
1548         // increment or decrement
1549         s := &ast.IncDecStmt{X: x[0], TokPos: p.pos,
1550         p.next()
1551         return s, false
1552     }
1553
1554     // expression
1555     return &ast.ExprStmt{X: x[0]}, false
1556 }
1557
1558 func (p *parser) parseCallExpr() *ast.CallExpr {
1559     x := p.parseRhsOrType() // could be a conversion: (s
1560     if call, isCall := x.(*ast.CallExpr); isCall {
1561         return call
1562     }
1563     if _, isBad := x.(*ast.BadExpr); !isBad {
1564         // only report error if it's a new one
1565         p.errorExpected(x.Pos(), "function/method ca
1566     }
1567     return nil
1568 }
1569
1570 func (p *parser) parseGoStmt() ast.Stmt {
1571     if p.trace {

```

```

1572         defer un(trace(p, "GoStmt"))
1573     }
1574
1575     pos := p.expect(token.GO)
1576     call := p.parseCallExpr()
1577     p.expectSemi()
1578     if call == nil {
1579         return &ast.BadStmt{From: pos, To: pos + 2}
1580     }
1581
1582     return &ast.GoStmt{Go: pos, Call: call}
1583 }
1584
1585 func (p *parser) parseDeferStmt() ast.Stmt {
1586     if p.trace {
1587         defer un(trace(p, "DeferStmt"))
1588     }
1589
1590     pos := p.expect(token.DEFER)
1591     call := p.parseCallExpr()
1592     p.expectSemi()
1593     if call == nil {
1594         return &ast.BadStmt{From: pos, To: pos + 5}
1595     }
1596
1597     return &ast.DeferStmt{Defer: pos, Call: call}
1598 }
1599
1600 func (p *parser) parseReturnStmt() *ast.ReturnStmt {
1601     if p.trace {
1602         defer un(trace(p, "ReturnStmt"))
1603     }
1604
1605     pos := p.pos
1606     p.expect(token.RETURN)
1607     var x []ast.Expr
1608     if p.tok != token.SEMICOLON && p.tok != token.RBRACE {
1609         x = p.parseRhsList()
1610     }
1611     p.expectSemi()
1612
1613     return &ast.ReturnStmt{Return: pos, Results: x}
1614 }
1615
1616 func (p *parser) parseBranchStmt(tok token.Token) *ast.BranchStmt {
1617     if p.trace {
1618         defer un(trace(p, "BranchStmt"))
1619     }
1620

```

```

1621     pos := p.expect(tok)
1622     var label *ast.Ident
1623     if tok != token.FALLTHROUGH && p.tok == token.IDENT
1624         label = p.parseIdent()
1625         // add to list of unresolved targets
1626         n := len(p.targetStack) - 1
1627         p.targetStack[n] = append(p.targetStack[n],
1628     }
1629     p.expectSemi()
1630
1631     return &ast.BranchStmt{TokPos: pos, Tok: tok, Label:
1632 }
1633
1634 func (p *parser) makeExpr(s ast.Stmt) ast.Expr {
1635     if s == nil {
1636         return nil
1637     }
1638     if es, isExpr := s.(*ast.ExprStmt); isExpr {
1639         return p.checkExpr(es.X)
1640     }
1641     p.error(s.Pos(), "expected condition, found simple s
1642     return &ast.BadExpr{From: s.Pos(), To: s.End()}
1643 }
1644
1645 func (p *parser) parseIfStmt() *ast.IfStmt {
1646     if p.trace {
1647         defer un(trace(p, "IfStmt"))
1648     }
1649
1650     pos := p.expect(token.IF)
1651     p.openScope()
1652     defer p.closeScope()
1653
1654     var s ast.Stmt
1655     var x ast.Expr
1656     {
1657         prevLev := p.exprLev
1658         p.exprLev = -1
1659         if p.tok == token.SEMICOLON {
1660             p.next()
1661             x = p.parseRhs()
1662         } else {
1663             s, _ = p.parseSimpleStmt(basic)
1664             if p.tok == token.SEMICOLON {
1665                 p.next()
1666                 x = p.parseRhs()
1667             } else {
1668                 x = p.makeExpr(s)
1669                 s = nil

```

```

1670         }
1671     }
1672     p.exprLev = prevLev
1673 }
1674
1675 body := p.parseBlockStmt()
1676 var else_ ast.Stmt
1677 if p.tok == token.ELSE {
1678     p.next()
1679     else_ = p.parseStmt()
1680 } else {
1681     p.expectSemi()
1682 }
1683
1684 return &ast.IfStmt{If: pos, Init: s, Cond: x, Body:
1685 }
1686
1687 func (p *parser) parseTypeList() (list []ast.Expr) {
1688     if p.trace {
1689         defer un(trace(p, "TypeList"))
1690     }
1691
1692     list = append(list, p.parseType())
1693     for p.tok == token.COMMA {
1694         p.next()
1695         list = append(list, p.parseType())
1696     }
1697
1698     return
1699 }
1700
1701 func (p *parser) parseCaseClause(typeSwitch bool) *ast.CaseC
1702     if p.trace {
1703         defer un(trace(p, "CaseClause"))
1704     }
1705
1706     pos := p.pos
1707     var list []ast.Expr
1708     if p.tok == token.CASE {
1709         p.next()
1710         if typeSwitch {
1711             list = p.parseTypeList()
1712         } else {
1713             list = p.parseRhsList()
1714         }
1715     } else {
1716         p.expect(token.DEFAULT)
1717     }
1718
1719     colon := p.expect(token.COLON)

```

```

1720         p.openScope()
1721         body := p.parseStmtList()
1722         p.closeScope()
1723
1724         return &ast.CaseClause{Case: pos, List: list, Colon:
1725     }
1726
1727 func isTypeSwitchAssert(x ast.Expr) bool {
1728     a, ok := x.(*ast.TypeAssertExpr)
1729     return ok && a.Type == nil
1730 }
1731
1732 func isTypeSwitchGuard(s ast.Stmt) bool {
1733     switch t := s.(type) {
1734     case *ast.ExprStmt:
1735         // x.(nil)
1736         return isTypeSwitchAssert(t.X)
1737     case *ast.AssignStmt:
1738         // v := x.(nil)
1739         return len(t.Lhs) == 1 && t.Tok == token.DEF
1740     }
1741     return false
1742 }
1743
1744 func (p *parser) parseSwitchStmt() ast.Stmt {
1745     if p.trace {
1746         defer un(trace(p, "SwitchStmt"))
1747     }
1748
1749     pos := p.expect(token.SWITCH)
1750     p.openScope()
1751     defer p.closeScope()
1752
1753     var s1, s2 ast.Stmt
1754     if p.tok != token.LBRACE {
1755         prevLev := p.exprLev
1756         p.exprLev = -1
1757         if p.tok != token.SEMICOLON {
1758             s2, _ = p.parseSimpleStmt(basic)
1759         }
1760         if p.tok == token.SEMICOLON {
1761             p.next()
1762             s1 = s2
1763             s2 = nil
1764             if p.tok != token.LBRACE {
1765                 // A TypeSwitchGuard may dec
1766                 // to the variable declared
1767                 // Introduce extra scope to
1768                 //

```

```

1769             //      switch t := 0; t :=
1770             //
1771             // (this code is not valid G
1772             // cannot be accessed and th
1773             // scope is needed for the c
1774             //
1775             // If we don't have a type s
1776             // Having the extra nested b
1777             p.openScope()
1778             defer p.closeScope()
1779             s2, _ = p.parseSimpleStmt(ba
1780         }
1781     }
1782     p.exprLev = prevLev
1783 }
1784
1785 typeSwitch := isTypeSwitchGuard(s2)
1786 lbrace := p.expect(token.LBRACE)
1787 var list []ast.Stmt
1788 for p.tok == token.CASE || p.tok == token.DEFAULT {
1789     list = append(list, p.parseCaseClause(typeSw
1790 }
1791 rbrace := p.expect(token.RBRACE)
1792 p.expectSemi()
1793 body := &ast.BlockStmt{Lbrace: lbrace, List: list, R
1794
1795 if typeSwitch {
1796     return &ast.TypeSwitchStmt{Switch: pos, Init
1797 }
1798
1799 return &ast.SwitchStmt{Switch: pos, Init: s1, Tag: p
1800 }
1801
1802 func (p *parser) parseCommClause() *ast.CommClause {
1803     if p.trace {
1804         defer un(trace(p, "CommClause"))
1805     }
1806
1807     p.openScope()
1808     pos := p.pos
1809     var comm ast.Stmt
1810     if p.tok == token.CASE {
1811         p.next()
1812         lhs := p.parseLhsList()
1813         if p.tok == token.ARROW {
1814             // SendStmt
1815             if len(lhs) > 1 {
1816                 p.errorExpected(lhs[0].Pos())
1817                 // continue with first expr

```

```

1818         }
1819         arrow := p.pos
1820         p.next()
1821         rhs := p.parseRhs()
1822         comm = &ast.SendStmt{Chan: lhs[0], A
1823     } else {
1824         // RecvStmt
1825         if tok := p.tok; tok == token.ASSIGN
1826             // RecvStmt with assignment
1827             if len(lhs) > 2 {
1828                 p.errorExpected(lhs[
1829                 // continue with fir
1830                 lhs = lhs[0:2]
1831             }
1832             pos := p.pos
1833             p.next()
1834             rhs := p.parseRhs()
1835             as := &ast.AssignStmt{Lhs: 1
1836             if tok == token.DEFINE {
1837                 p.shortVarDecl(as, 1
1838             }
1839             comm = as
1840         } else {
1841             // lhs must be single receiv
1842             if len(lhs) > 1 {
1843                 p.errorExpected(lhs[
1844                 // continue with fir
1845             }
1846             comm = &ast.ExprStmt{X: lhs[
1847         }
1848     }
1849 } else {
1850     p.expect(token.DEFAULT)
1851 }
1852
1853 colon := p.expect(token.COLON)
1854 body := p.parseStmtList()
1855 p.closeScope()
1856
1857 return &ast.CommClause{Case: pos, Comm: comm, Colon:
1858 }
1859
1860 func (p *parser) parseSelectStmt() *ast.SelectStmt {
1861     if p.trace {
1862         defer un(trace(p, "SelectStmt"))
1863     }
1864
1865     pos := p.expect(token.SELECT)
1866     lbrace := p.expect(token.LBRACE)
1867     var list []ast.Stmt

```

```

1868         for p.tok == token.CASE || p.tok == token.DEFAULT {
1869             list = append(list, p.parseCommClause())
1870         }
1871         rbrace := p.expect(token.RBRACE)
1872         p.expectSemi()
1873         body := &ast.BlockStmt{Lbrace: lbrace, List: list, R
1874
1875             return &ast.SelectStmt{Select: pos, Body: body}
1876     }
1877
1878     func (p *parser) parseForStmt() ast.Stmt {
1879         if p.trace {
1880             defer un(trace(p, "ForStmt"))
1881         }
1882
1883         pos := p.expect(token.FOR)
1884         p.openScope()
1885         defer p.closeScope()
1886
1887         var s1, s2, s3 ast.Stmt
1888         var isRange bool
1889         if p.tok != token.LBRACE {
1890             prevLev := p.exprLev
1891             p.exprLev = -1
1892             if p.tok != token.SEMICOLON {
1893                 s2, isRange = p.parseSimpleStmt(rang
1894             }
1895             if !isRange && p.tok == token.SEMICOLON {
1896                 p.next()
1897                 s1 = s2
1898                 s2 = nil
1899                 if p.tok != token.SEMICOLON {
1900                     s2, _ = p.parseSimpleStmt(ba
1901                 }
1902                 p.expectSemi()
1903                 if p.tok != token.LBRACE {
1904                     s3, _ = p.parseSimpleStmt(ba
1905                 }
1906             }
1907             p.exprLev = prevLev
1908         }
1909
1910         body := p.parseBlockStmt()
1911         p.expectSemi()
1912
1913         if isRange {
1914             as := s2.(*ast.AssignStmt)
1915             // check lhs
1916             var key, value ast.Expr

```

```

1917         switch len(as.Lhs) {
1918         case 2:
1919             key, value = as.Lhs[0], as.Lhs[1]
1920         case 1:
1921             key = as.Lhs[0]
1922         default:
1923             p.errorExpected(as.Lhs[0].Pos(), "1
1924             return &ast.BadStmt{From: pos, To: b
1925         }
1926         // parseSimpleStmt returned a right-hand sid
1927         // is a single unary expression of the form
1928         x := as.Rhs[0].(*ast.UnaryExpr).X
1929         return &ast.RangeStmt{
1930             For:    pos,
1931             Key:    key,
1932             Value:  value,
1933             TokPos: as.TokPos,
1934             Tok:    as.Tok,
1935             X:      x,
1936             Body:  body,
1937         }
1938     }
1939
1940     // regular for statement
1941     return &ast.ForStmt{
1942         For:  pos,
1943         Init: s1,
1944         Cond: p.makeExpr(s2),
1945         Post: s3,
1946         Body: body,
1947     }
1948 }
1949
1950 func (p *parser) parseStmt() (s ast.Stmt) {
1951     if p.trace {
1952         defer un(trace(p, "Statement"))
1953     }
1954
1955     switch p.tok {
1956     case token.CONST, token.TYPE, token.VAR:
1957         s = &ast.DeclStmt{Decl: p.parseDecl(syncStmt
1958     case
1959         // tokens that may start an expression
1960         token.IDENT, token.INT, token.FLOAT, token.I
1961         token.LBRACK, token.STRUCT, // composite typ
1962         token.ADD, token.SUB, token.MUL, token.AND,
1963         s, _ = p.parseSimpleStmt(labelOk)
1964         // because of the required look-ahead, label
1965         // parsed by parseSimpleStmt - don't expect

```

```

1966             // them
1967             if _, isLabeledStmt := s.(*ast.LabeledStmt);
1968                 p.expectSemi()
1969             }
1970     case token.GO:
1971         s = p.parseGoStmt()
1972     case token.DEFER:
1973         s = p.parseDeferStmt()
1974     case token.RETURN:
1975         s = p.parseReturnStmt()
1976     case token.BREAK, token.CONTINUE, token.GOTO, token.
1977         s = p.parseBranchStmt(p.tok)
1978     case token.LBRACE:
1979         s = p.parseBlockStmt()
1980         p.expectSemi()
1981     case token.IF:
1982         s = p.parseIfStmt()
1983     case token.SWITCH:
1984         s = p.parseSwitchStmt()
1985     case token.SELECT:
1986         s = p.parseSelectStmt()
1987     case token.FOR:
1988         s = p.parseForStmt()
1989     case token.SEMICOLON:
1990         s = &ast.EmptyStmt{Semicolon: p.pos}
1991         p.next()
1992     case token.RBRACE:
1993         // a semicolon may be omitted before a closi
1994         s = &ast.EmptyStmt{Semicolon: p.pos}
1995     default:
1996         // no statement found
1997         pos := p.pos
1998         p.errorExpected(pos, "statement")
1999         syncStmt(p)
2000         s = &ast.BadStmt{From: pos, To: p.pos}
2001     }
2002
2003     return
2004 }
2005
2006 // -----
2007 // Declarations
2008
2009 type parseSpecFunction func(p *parser, doc *ast.CommentGroup
2010
2011 func isValidImport(lit string) bool {
2012     const illegalChars = `!#$%&'()*,:;<=>?[\]^_{|}` + "`
2013     s, _ := strconv.Unquote(lit) // go/scanner returns a
2014     for _, r := range s {
2015         if !unicode.IsGraphic(r) || unicode.IsSpace(

```

```

2016             return false
2017         }
2018     }
2019     return s != ""
2020 }
2021
2022 func parseImportSpec(p *parser, doc *ast.CommentGroup, _ int
2023     if p.trace {
2024         defer un(trace(p, "ImportSpec"))
2025     }
2026
2027     var ident *ast.Ident
2028     switch p.tok {
2029     case token.PERIOD:
2030         ident = &ast.Ident{NamePos: p.pos, Name: "."
2031             p.next()
2032     case token.IDENT:
2033         ident = p.parseIdent()
2034     }
2035
2036     var path *ast.BasicLit
2037     if p.tok == token.STRING {
2038         if !isValidImport(p.lit) {
2039             p.error(p.pos, "invalid import path:
2040         }
2041         path = &ast.BasicLit{ValuePos: p.pos, Kind:
2042             p.next()
2043     } else {
2044         p.expect(token.STRING) // use expect() error
2045     }
2046     p.expectSemi() // call before accessing p.linecommen
2047
2048     // collect imports
2049     spec := &ast.ImportSpec{
2050         Doc:     doc,
2051         Name:     ident,
2052         Path:     path,
2053         Comment: p.lineComment,
2054     }
2055     p.imports = append(p.imports, spec)
2056
2057     return spec
2058 }
2059
2060 func parseConstSpec(p *parser, doc *ast.CommentGroup, iota i
2061     if p.trace {
2062         defer un(trace(p, "ConstSpec"))
2063     }
2064

```

```

2065         idents := p.parseIdentList()
2066         typ := p.tryType()
2067         var values []ast.Expr
2068         if typ != nil || p.tok == token.ASSIGN || iota == 0
2069             p.expect(token.ASSIGN)
2070             values = p.parseRhsList()
2071     }
2072     p.expectSemi() // call before accessing p.linecommen
2073
2074     // Go spec: The scope of a constant or variable iden
2075     // a function begins at the end of the ConstSpec or
2076     // the end of the innermost containing block.
2077     // (Global identifiers are resolved in a separate ph
2078     spec := &ast.ValueSpec{
2079         Doc:      doc,
2080         Names:    idents,
2081         Type:     typ,
2082         Values:   values,
2083         Comment:  p.lineComment,
2084     }
2085     p.declare(spec, iota, p.topScope, ast.Con, idents...
2086
2087     return spec
2088 }
2089
2090 func parseTypeSpec(p *parser, doc *ast.CommentGroup, _ int)
2091     if p.trace {
2092         defer un(trace(p, "TypeSpec"))
2093     }
2094
2095     ident := p.parseIdent()
2096
2097     // Go spec: The scope of a type identifier declared
2098     // at the identifier in the TypeSpec and ends at the
2099     // containing block.
2100     // (Global identifiers are resolved in a separate ph
2101     spec := &ast.TypeSpec{Doc: doc, Name: ident}
2102     p.declare(spec, nil, p.topScope, ast.Typ, ident)
2103
2104     spec.Type = p.parseType()
2105     p.expectSemi() // call before accessing p.linecommen
2106     spec.Comment = p.lineComment
2107
2108     return spec
2109 }
2110
2111 func parseVarSpec(p *parser, doc *ast.CommentGroup, _ int) a
2112     if p.trace {
2113         defer un(trace(p, "VarSpec"))

```

```

2114     }
2115
2116     idents := p.parseIdentList()
2117     typ := p.tryType()
2118     var values []ast.Expr
2119     if typ == nil || p.tok == token.ASSIGN {
2120         p.expect(token.ASSIGN)
2121         values = p.parseRhsList()
2122     }
2123     p.expectSemi() // call before accessing p.linecommen
2124
2125     // Go spec: The scope of a constant or variable iden
2126     // a function begins at the end of the ConstSpec or
2127     // the end of the innermost containing block.
2128     // (Global identifiers are resolved in a separate ph
2129     spec := &ast.ValueSpec{
2130         Doc:     doc,
2131         Names:   idents,
2132         Type:    typ,
2133         Values:  values,
2134         Comment: p.lineComment,
2135     }
2136     p.declare(spec, nil, p.topScope, ast.Var, idents...)
2137
2138     return spec
2139 }
2140
2141 func (p *parser) parseGenDecl(keyword token.Token, f parseSp
2142     if p.trace {
2143         defer un(trace(p, "GenDecl("+keyword.String(
2144     }
2145
2146     doc := p.leadComment
2147     pos := p.expect(keyword)
2148     var lparen, rparen token.Pos
2149     var list []ast.Spec
2150     if p.tok == token.LPAREN {
2151         lparen = p.pos
2152         p.next()
2153         for iota := 0; p.tok != token.RPAREN && p.to
2154             list = append(list, f(p, p.leadComme
2155     }
2156     rparen = p.expect(token.RPAREN)
2157     p.expectSemi()
2158 } else {
2159     list = append(list, f(p, nil, 0))
2160 }
2161
2162     return &ast.GenDecl{
2163         Doc:     doc,

```

```

2164         TokPos: pos,
2165         Tok:    keyword,
2166         Lparen: lparen,
2167         Specs:  list,
2168         Rparen: rparen,
2169     }
2170 }
2171
2172 func (p *parser) parseReceiver(scope *ast.Scope) *ast.FieldL
2173     if p.trace {
2174         defer un(trace(p, "Receiver"))
2175     }
2176
2177     par := p.parseParameters(scope, false)
2178
2179     // must have exactly one receiver
2180     if par.NumFields() != 1 {
2181         p.errorExpected(par.Opening, "exactly one re
2182         par.List = []*ast.Field{{Type: &ast.BadExpr{
2183         return par
2184     }
2185
2186     // recv type must be of the form ["*"] identifier
2187     recv := par.List[0]
2188     base := deref(recv.Type)
2189     if _, isIdent := base.(*ast.Ident); !isIdent {
2190         if _, isBad := base.(*ast.BadExpr); !isBad {
2191             // only report error if it's a new o
2192             p.errorExpected(base.Pos(), "(unqual
2193         }
2194         par.List = []*ast.Field{
2195             {Type: &ast.BadExpr{From: recv.Pos()
2196         }
2197     }
2198
2199     return par
2200 }
2201
2202 func (p *parser) parseFuncDecl() *ast.FuncDecl {
2203     if p.trace {
2204         defer un(trace(p, "FunctionDecl"))
2205     }
2206
2207     doc := p.leadComment
2208     pos := p.expect(token.FUNC)
2209     scope := ast.NewScope(p.topScope) // function scope
2210
2211     var recv *ast.FieldList
2212     if p.tok == token.LPAREN {

```

```

2213         recv = p.parseReceiver(scope)
2214     }
2215
2216     ident := p.parseIdent()
2217
2218     params, results := p.parseSignature(scope)
2219
2220     var body *ast.BlockStmt
2221     if p.tok == token.LBRACE {
2222         body = p.parseBody(scope)
2223     }
2224     p.expectSemi()
2225
2226     decl := &ast.FuncDecl{
2227         Doc:  doc,
2228         Recv: recv,
2229         Name: ident,
2230         Type: &ast.FuncType{
2231             Func:  pos,
2232             Params: params,
2233             Results: results,
2234         },
2235         Body: body,
2236     }
2237     if recv == nil {
2238         // Go spec: The scope of an identifier denot
2239         // variable, or function (but not method) de
2240         // (outside any function) is the package blo
2241         //
2242         // init() functions cannot be referred to an
2243         // be more than one - don't put them in the
2244         if ident.Name != "init" {
2245             p.declare(decl, nil, p.pkgScope, ast
2246         }
2247     }
2248
2249     return decl
2250 }
2251
2252 func (p *parser) parseDecl(sync func(*parser)) ast.Decl {
2253     if p.trace {
2254         defer un(trace(p, "Declaration"))
2255     }
2256
2257     var f parseSpecFunction
2258     switch p.tok {
2259     case token.CONST:
2260         f = parseConstSpec
2261

```

```

2262     case token.TYPE:
2263         f = parseTypeSpec
2264
2265     case token.VAR:
2266         f = parseVarSpec
2267
2268     case token.FUNC:
2269         return p.parseFuncDecl()
2270
2271     default:
2272         pos := p.pos
2273         p.errorExpected(pos, "declaration")
2274         sync(p)
2275         return &ast.BadDecl{From: pos, To: p.pos}
2276     }
2277
2278     return p.parseGenDecl(p.tok, f)
2279 }
2280
2281 // -----
2282 // Source files
2283
2284 func (p *parser) parseFile() *ast.File {
2285     if p.trace {
2286         defer un(trace(p, "File"))
2287     }
2288
2289     // package clause
2290     doc := p.leadComment
2291     pos := p.expect(token.PACKAGE)
2292     // Go spec: The package clause is not a declaration;
2293     // the package name does not appear in any scope.
2294     ident := p.parseIdent()
2295     if ident.Name == "_" {
2296         p.error(p.pos, "invalid package name _")
2297     }
2298     p.expectSemi()
2299
2300     var decls []ast.Decl
2301
2302     // Don't bother parsing the rest if we had errors al
2303     // Likely not a Go source file at all.
2304
2305     if p.errors.Len() == 0 && p.mode&PackageClauseOnly =
2306         // import decls
2307         for p.tok == token.IMPORT {
2308             decls = append(decls, p.parseGenDecl
2309         }
2310
2311     if p.mode&ImportsOnly == 0 {

```

```

2312             // rest of package body
2313             for p.tok != token.EOF {
2314                 decls = append(decls, p.pars
2315             }
2316         }
2317     }
2318
2319     assert(p.topScope == p.pkgScope, "imbalanced scopes"
2320
2321     // resolve global identifiers within the same file
2322     i := 0
2323     for _, ident := range p.unresolved {
2324         // i <= index for current ident
2325         assert(ident.Obj == unresolved, "object already
2326         ident.Obj = p.pkgScope.Lookup(ident.Name) //
2327         if ident.Obj == nil {
2328             p.unresolved[i] = ident
2329             i++
2330         }
2331     }
2332
2333     return &ast.File{
2334         Doc:      doc,
2335         Package:   pos,
2336         Name:      ident,
2337         Decls:     decls,
2338         Scope:     p.pkgScope,
2339         Imports:   p.imports,
2340         Unresolved: p.unresolved[0:i],
2341         Comments:  p.comments,
2342     }
2343 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/go/printer/nodes.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file implements printing of AST nodes; specifically
6 // expressions, statements, declarations, and files. It uses
7 // the print functionality implemented in printer.go.
8
9 package printer
10
11 import (
12     "bytes"
13     "go/ast"
14     "go/token"
15     "unicode/utf8"
16 )
17
18 // Formatting issues:
19 // - better comment formatting for /*-style comments at the
20 //   when the comment spans multiple lines; if such a commen
21 //   not idempotent
22 // - formatting of expression lists
23 // - should use blank instead of tab to separate one-line fu
24 //   the function header unless there is a group of consecut
25
26 // -----
27 // Common AST nodes.
28
29 // Print as many newlines as necessary (but at least min new
30 // the current line. ws is printed before the first line bre
31 // is set, the first line break is printed as formfeed. Retu
32 // line break was printed; returns false otherwise.
33 //
34 // TODO(gri): linebreak may add too many lines if the next s
35 //             is preceded by comments because the computatio
36 //             the current position before the comment and th
37 //             after the comment. Thus, after interspersing s
38 //             space taken up by them is not considered to re
39 //             linebreaks. At the moment there is no easy way
40 //             future (not yet interspersed) comments in this
41 //
```

```

42 func (p *printer) linebreak(line, min int, ws whitespace, ne
43     n := nlimit(line - p.pos.Line)
44     if n < min {
45         n = min
46     }
47     if n > 0 {
48         p.print(ws)
49         if newSection {
50             p.print(formfeed)
51             n--
52         }
53         for ; n > 0; n-- {
54             p.print(newline)
55         }
56         printedBreak = true
57     }
58     return
59 }
60
61 // setComment sets g as the next comment if g != nil and if
62 // are enabled - this mode is used when printing source code
63 // as exports only. It assumes that there are no other pendi
64 // intersperse.
65 func (p *printer) setComment(g *ast.CommentGroup) {
66     if g == nil || !p.useNodeComments {
67         return
68     }
69     if p.comments == nil {
70         // initialize p.comments lazily
71         p.comments = make([]*ast.CommentGroup, 1)
72     } else if p.cindex < len(p.comments) {
73         // for some reason there are pending comment
74         // should never happen - handle gracefully a
75         // all comments up to g, ignore anything aft
76         p.flush(p.posFor(g.List[0].Pos()), token.ILL
77     }
78     p.comments[0] = g
79     p.cindex = 0
80     p.nextComment() // get comment ready for use
81 }
82
83 type exprListMode uint
84
85 const (
86     commaTerm exprListMode = 1 << iota // list is option
87     noIndent                          // no extra inden
88 )
89
90 // If indent is set, a multi-line identifier list is indente
91 // first linebreak encountered.

```

```

92 func (p *printer) identList(list []*ast.Ident, indent bool)
93 // convert into an expression list so we can re-use
94 xlist := make([]*ast.Expr, len(list))
95 for i, x := range list {
96     xlist[i] = x
97 }
98 var mode exprListMode
99 if !indent {
100     mode = noIndent
101 }
102 p.exprList(token.NoPos, xlist, 1, mode, token.NoPos)
103 }
104
105 // Print a list of expressions. If the list spans multiple
106 // source lines, the original line breaks are respected betw
107 // expressions.
108 //
109 // TODO(gri) Consider rewriting this to be independent of []
110 //           so that we can use the algorithm for any kind o
111 //           (e.g., pass list via a channel over which to ra
112 func (p *printer) exprList(prev token.Pos, list []*ast.Expr,
113     if len(list) == 0 {
114         return
115     }
116
117     prev := p.posFor(prev)
118     next := p.posFor(next)
119     line := p.lineFor(list[0].Pos())
120     endLine := p.lineFor(list[len(list)-1].End())
121
122     if prev.IsValid() && prev.Line == line && line == en
123         // all list entries on a single line
124         for i, x := range list {
125             if i > 0 {
126                 // use position of expressio
127                 // comma position for correc
128                 p.print(x.Pos(), token.COMMA
129             }
130             p.expr(x, depth)
131         }
132         return
133     }
134
135     // list entries span multiple lines;
136     // use source code positions to guide line breaks
137
138     // don't add extra indentation if noIndent is set;
139     // i.e., pretend that the first line is already inde
140     ws := ignore

```

```

141     if mode&noIndent == 0 {
142         ws = indent
143     }
144
145     // the first linebreak is always a formfeed since th
146     // depend on any previous formatting
147     prevBreak := -1 // index of last expression that was
148     if prev.IsValid() && prev.Line < line && p.linebreak
149         ws = ignore
150         prevBreak = 0
151     }
152
153     // initialize expression/key size: a zero value indi
154     size := 0
155
156     // print all list elements
157     for i, x := range list {
158         prevLine := line
159         line = p.lineFor(x.Pos())
160
161         // determine if the next linebreak, if any,
162         // in general, use the entire node size to m
163         // key:value expressions, use the key size
164         // TODO(gri) for a better result, should pro
165         // the key and the node size into
166         useFF := true
167
168         // determine element size: all bets are off
169         // position information for the previous and
170         // generated code - simply ignore the size i
171         // it to 0)
172         prevSize := size
173         const infinity = 1e6 // larger than any sour
174         size = p.nodeSize(x, infinity)
175         pair, isPair := x.(*ast.KeyValueExpr)
176         if size <= infinity && prev.IsValid() && nex
177             // x fits on a single line
178             if isPair {
179                 size = p.nodeSize(pair.Key,
180             }
181         } else {
182             // size too large or we don't have g
183             size = 0
184         }
185
186         // if the previous line and the current line
187         // line-expressions and the key sizes are sm
188         // the ratio between the key sizes does not
189         // threshold, align columns and do not use f

```

```

190         if prevSize > 0 && size > 0 {
191             const smallSize = 20
192             if prevSize <= smallSize && size <=
193                 useFF = false
194             } else {
195                 const r = 4 // threshold
196                 ratio := float64(size) / flo
197                 useFF = ratio <= 1/r || r <=
198             }
199         }
200
201     if i > 0 {
202         needsLinebreak := prevLine < line &&
203             // use position of expression follow
204             // comma position for correct commen
205             // only if the expression is on the
206             if !needsLinebreak {
207                 p.print(x.Pos())
208             }
209             p.print(token.COMMA)
210             needsBlank := true
211             if needsLinebreak {
212                 // lines are broken using ne
213                 // unless forceFF is set or
214                 // the same line in which ca
215                 if p.linebreak(line, 0, ws,
216                     ws = ignore
217                     prevBreak = i
218                     needsBlank = false /
219                 }
220             }
221             if needsBlank {
222                 p.print(blank)
223             }
224         }
225
226     if isPair && size > 0 && len(list) > 1 {
227         // we have a key:value expression th
228         // is in a list with more then one e
229         // key such that consecutive entries
230         p.expr(pair.Key)
231         p.print(pair.Colon, token.COLON, vta
232         p.expr(pair.Value)
233     } else {
234         p.expr0(x, depth)
235     }
236 }
237
238 if mode&commaTerm != 0 && next.IsValid() && p.pos.Li
239     // print a terminating comma if the next tok

```

```

240         p.print(token.COMMA)
241         if ws == ignore && mode&noIndent == 0 {
242             // unindent if we indented
243             p.print(unindent)
244         }
245         p.print(formfeed) // terminating comma needs
246         return
247     }
248
249     if ws == ignore && mode&noIndent == 0 {
250         // unindent if we indented
251         p.print(unindent)
252     }
253 }
254
255 func (p *printer) parameters(fields *ast.FieldList) {
256     p.print(fields.Opening, token.LPAREN)
257     if len(fields.List) > 0 {
258         prevLine := p.lineFor(fields.Opening)
259         ws := indent
260         for i, par := range fields.List {
261             // determine par begin and end line
262             // if there are multiple parameter n
263             // or the type is on a separate line
264             var parLineBeg int
265             var parLineEnd = p.lineFor(par.Type)
266             if len(par.Names) > 0 {
267                 parLineBeg = p.lineFor(par.N
268             } else {
269                 parLineBeg = parLineEnd
270             }
271             // separating "," if needed
272             needsLinebreak := 0 < prevLine && pr
273             if i > 0 {
274                 // use position of parameter
275                 // comma position for correc
276                 // only if the next paramete
277                 if !needsLinebreak {
278                     p.print(par.Pos())
279                 }
280                 p.print(token.COMMA)
281             }
282             // separator if needed (linebreak or
283             if needsLinebreak && p.linebreak(par
284                 // break line if the opening
285                 ws = ignore
286             } else if i > 0 {
287                 p.print(blank)
288             }

```

```

289         // parameter names
290         if len(par.Names) > 0 {
291             // Very subtle: If we indent
292             // won't indent again. If we
293             // indent if the identList s
294             // again at the end (and sti
295             // by a linebreak call after
296             // will do the right thing.
297             p.identList(par.Names, ws ==
298             p.print(blank)
299         }
300         // parameter type
301         p.expr(par.Type)
302         prevLine = parLineEnd
303     }
304     // if the closing ")" is on a separate line
305     // print an additional "," and line break
306     if closing := p.lineFor(fields.Closing); 0 <
307         p.print(token.COMMA)
308         p.linebreak(closing, 0, ignore, true
309     }
310     // unindent if we indented
311     if ws == ignore {
312         p.print(unindent)
313     }
314 }
315 p.print(fields.Closing, token.RPAREN)
316 }
317
318 func (p *printer) signature(params, result *ast.FieldList) {
319     p.parameters(params)
320     n := result.NumFields()
321     if n > 0 {
322         p.print(blank)
323         if n == 1 && result.List[0].Names == nil {
324             // single anonymous result; no ()'s
325             p.expr(result.List[0].Type)
326             return
327         }
328         p.parameters(result)
329     }
330 }
331
332 func identListSize(list []*ast.Ident, maxSize int) (size int)
333     for i, x := range list {
334         if i > 0 {
335             size += len(", ")
336         }
337         size += utf8.RuneCountInString(x.Name)

```

```

338         if size >= maxSize {
339             break
340         }
341     }
342     return
343 }
344
345 func (p *printer) isOneLineFieldList(list []*ast.Field) bool
346     if len(list) != 1 {
347         return false // allow only one field
348     }
349     f := list[0]
350     if f.Tag != nil || f.Comment != nil {
351         return false // don't allow tags or comments
352     }
353     // only name(s) and type
354     const maxSize = 30 // adjust as appropriate, this is
355     namesSize := identListSize(f.Names, maxSize)
356     if namesSize > 0 {
357         namesSize = 1 // blank between names and typ
358     }
359     typeSize := p.nodeSize(f.Type, maxSize)
360     return namesSize+typeSize <= maxSize
361 }
362
363 func (p *printer) setLineComment(text string) {
364     p.setComment(&ast.CommentGroup{List: []*ast.Comment{
365     }
366 }
367
368 func (p *printer) isMultiLine(n ast.Node) bool {
369     return p.lineFor(n.End())-p.lineFor(n.Pos()) > 0
370 }
371
372 func (p *printer) fieldList(fields *ast.FieldList, isStruct,
373     lbrace := fields.Opening
374     list := fields.List
375     rbrace := fields.Closing
376     hasComments := isIncomplete || p.commentBefore(p.pos
377     srcIsOneLine := lbrace.IsValid() && rbrace.IsValid()
378
379     if !hasComments && srcIsOneLine {
380         // possibly a one-line struct/interface
381         if len(list) == 0 {
382             // no blank between keyword and {} i
383             p.print(lbrace, token.LBRACE, rbrace
384             return
385         } else if isStruct && p.isOneLineFieldList(l
386         // small enough - print on one line
387         // (don't use identList and ignore s
388         p.print(lbrace, token.LBRACE, blank)

```

```

388         f := list[0]
389         for i, x := range f.Names {
390             if i > 0 {
391                 // no comments so no
392                 p.print(token.COMMA,
393                     }
394                 p.expr(x)
395             }
396             if len(f.Names) > 0 {
397                 p.print(blank)
398             }
399             p.expr(f.Type)
400             p.print(blank, rbrace, token.RBRACE)
401             return
402         }
403     }
404     // hasComments || !srcIsOneLine
405
406     p.print(blank, lbrace, token.LBRACE, indent)
407     if hasComments || len(list) > 0 {
408         p.print(formfeed)
409     }
410
411     if isStruct {
412
413         sep := vtab
414         if len(list) == 1 {
415             sep = blank
416         }
417         newSection := false
418         for i, f := range list {
419             if i > 0 {
420                 p.linebreak(p.lineFor(f.Pos(
421                     }
422                 extraTabs := 0
423                 p.setComment(f.Doc)
424                 if len(f.Names) > 0 {
425                     // named fields
426                     p.identList(f.Names, false)
427                     p.print(sep)
428                     p.expr(f.Type)
429                     extraTabs = 1
430                 } else {
431                     // anonymous field
432                     p.expr(f.Type)
433                     extraTabs = 2
434                 }
435                 if f.Tag != nil {
436                     if len(f.Names) > 0 && sep =

```

```

437             p.print(sep)
438         }
439         p.print(sep)
440         p.expr(f.Tag)
441         extraTabs = 0
442     }
443     if f.Comment != nil {
444         for ; extraTabs > 0; extraTa
445             p.print(sep)
446         }
447         p.setComment(f.Comment)
448     }
449     newSection = p.isMultiLine(f)
450 }
451 if isIncomplete {
452     if len(list) > 0 {
453         p.print(formfeed)
454     }
455     p.flush(p.posFor(rbrace), token.RBRA
456     p.setLineComment("// contains filter
457 }
458
459 } else { // interface
460
461     newSection := false
462     for i, f := range list {
463         if i > 0 {
464             p.linebreak(p.lineFor(f.Pos(
465         }
466         p.setComment(f.Doc)
467         if ftyp, isFtyp := f.Type.(*ast.Func
468             // method
469             p.expr(f.Names[0])
470             p.signature(ftyp.Params, fty
471         } else {
472             // embedded interface
473             p.expr(f.Type)
474         }
475         p.setComment(f.Comment)
476         newSection = p.isMultiLine(f)
477     }
478     if isIncomplete {
479         if len(list) > 0 {
480             p.print(formfeed)
481         }
482         p.flush(p.posFor(rbrace), token.RBRA
483         p.setLineComment("// contains filter
484     }
485

```

```

486     }
487     p.print(unindent, formfeed, rbrace, token.RBRACE)
488 }
489
490 // -----
491 // Expressions
492
493 func walkBinary(e *ast.BinaryExpr) (has4, has5 bool, maxProb
494     switch e.Op.Precedence() {
495     case 4:
496         has4 = true
497     case 5:
498         has5 = true
499     }
500
501     switch l := e.X.(type) {
502     case *ast.BinaryExpr:
503         if l.Op.Precedence() < e.Op.Precedence() {
504             // parens will be inserted.
505             // pretend this is an *ast.ParenExpr
506             break
507         }
508         h4, h5, mp := walkBinary(l)
509         has4 = has4 || h4
510         has5 = has5 || h5
511         if maxProblem < mp {
512             maxProblem = mp
513         }
514     }
515
516     switch r := e.Y.(type) {
517     case *ast.BinaryExpr:
518         if r.Op.Precedence() <= e.Op.Precedence() {
519             // parens will be inserted.
520             // pretend this is an *ast.ParenExpr
521             break
522         }
523         h4, h5, mp := walkBinary(r)
524         has4 = has4 || h4
525         has5 = has5 || h5
526         if maxProblem < mp {
527             maxProblem = mp
528         }
529
530     case *ast.StarExpr:
531         if e.Op == token.QUO { // `*/`
532             maxProblem = 5
533         }
534
535     case *ast.UnaryExpr:

```

```

536         switch e.Op.String() + r.Op.String() {
537         case "/*", "&&", "&^":
538             maxProblem = 5
539         case "++", "--":
540             if maxProblem < 4 {
541                 maxProblem = 4
542             }
543         }
544     }
545     return
546 }
547
548 func cutoff(e *ast.BinaryExpr, depth int) int {
549     has4, has5, maxProblem := walkBinary(e)
550     if maxProblem > 0 {
551         return maxProblem + 1
552     }
553     if has4 && has5 {
554         if depth == 1 {
555             return 5
556         }
557         return 4
558     }
559     if depth == 1 {
560         return 6
561     }
562     return 4
563 }
564
565 func diffPrec(expr ast.Expr, prec int) int {
566     x, ok := expr.(*ast.BinaryExpr)
567     if !ok || prec != x.Op.Precedence() {
568         return 1
569     }
570     return 0
571 }
572
573 func reduceDepth(depth int) int {
574     depth--
575     if depth < 1 {
576         depth = 1
577     }
578     return depth
579 }
580
581 // Format the binary expression: decide the cutoff and then
582 // Let's call depth == 1 Normal mode, and depth > 1 Compact
583 // (Algorithm suggestion by Russ Cox.)
584 //

```

```

585 // The precedences are:
586 //     5         *  /  %  <<  >>  &  &^
587 //     4         +  -  |  ^
588 //     3         ==  !=  <  <=  >  >=
589 //     2         &&
590 //     1         ||
591 //
592 // The only decision is whether there will be spaces around
593 // There are never spaces at level 6 (unary), and always spa
594 //
595 // To choose the cutoff, look at the whole expression but ex
596 // expressions (function calls, parenthesized exprs), and ap
597 //
598 //     1) If there is a binary operator with a right side u
599 //         that would clash without a space, the cutoff must
600 //
601 //         /*         6
602 //         &&         6
603 //         &^        6
604 //         ++         5
605 //         --         5
606 //
607 //         (Comparison operators always have spaces around t
608 //
609 //     2) If there is a mix of level 5 and level 4 operator
610 //         is 5 (use spaces to distinguish precedence) in No
611 //         and 4 (never use spaces) in Compact mode.
612 //
613 //     3) If there are no level 4 operators or no level 5 o
614 //         cutoff is 6 (always use spaces) in Normal mode
615 //         and 4 (never use spaces) in Compact mode.
616 //
617 func (p *printer) binaryExpr(x *ast.BinaryExpr, prec1, cutoff
618     prec := x.Op.Precedence()
619     if prec < prec1 {
620         // parenthesis needed
621         // Note: The parser inserts an ast.ParenExpr
622         //         can only occur if the AST is create
623         p.print(token.LPAREN)
624         p.expr0(x, reduceDepth(depth)) // parenthese
625         p.print(token.RPAREN)
626         return
627     }
628
629     printBlank := prec < cutoff
630
631     ws := indent
632     p.expr1(x.X, prec, depth+diffPrec(x.X, prec))
633     if printBlank {

```

```

634         p.print(blank)
635     }
636     xline := p.pos.Line // before the operator (it may b
637     yline := p.lineFor(x.Y.Pos())
638     p.print(x.OpPos, x.Op)
639     if xline != yline && xline > 0 && yline > 0 {
640         // at least one line break, but respect an e
641         // in the source
642         if p.linebreak(yline, 1, ws, true) {
643             ws = ignore
644             printBlank = false // no blank after
645         }
646     }
647     if printBlank {
648         p.print(blank)
649     }
650     p.expr1(x.Y, prec+1, depth+1)
651     if ws == ignore {
652         p.print(unindent)
653     }
654 }
655
656 func isBinary(expr ast.Expr) bool {
657     _, ok := expr.(*ast.BinaryExpr)
658     return ok
659 }
660
661 func (p *printer) expr1(expr ast.Expr, prec1, depth int) {
662     p.print(expr.Pos())
663
664     switch x := expr.(type) {
665     case *ast.BadExpr:
666         p.print("BadExpr")
667
668     case *ast.Ident:
669         p.print(x)
670
671     case *ast.BinaryExpr:
672         if depth < 1 {
673             p.internalError("depth < 1:", depth)
674             depth = 1
675         }
676         p.binaryExpr(x, prec1, cutoff(x, depth), dep
677
678     case *ast.KeyValueExpr:
679         p.expr(x.Key)
680         p.print(x.Colon, token.COLON, blank)
681         p.expr(x.Value)
682
683     case *ast.StarExpr:

```

```

684         const prec = token.UnaryPrec
685         if prec < prec1 {
686             // parenthesis needed
687             p.print(token.LPAREN)
688             p.print(token.MUL)
689             p.expr(x.X)
690             p.print(token.RPAREN)
691         } else {
692             // no parenthesis needed
693             p.print(token.MUL)
694             p.expr(x.X)
695         }
696
697     case *ast.UnaryExpr:
698         const prec = token.UnaryPrec
699         if prec < prec1 {
700             // parenthesis needed
701             p.print(token.LPAREN)
702             p.expr(x)
703             p.print(token.RPAREN)
704         } else {
705             // no parenthesis needed
706             p.print(x.Op)
707             if x.Op == token.RANGE {
708                 // TODO(gri) Remove this cod
709                 p.print(blank)
710             }
711             p.expr1(x.X, prec, depth)
712         }
713
714     case *ast.BasicLit:
715         p.print(x)
716
717     case *ast.FuncLit:
718         p.expr(x.Type)
719         p.funcBody(x.Body, p.distance(x.Type.Pos()),
720
721     case *ast.ParenExpr:
722         if _, hasParens := x.X.(*ast.ParenExpr); has
723             // don't print parentheses around an
724             // TODO(gri) consider making this mo
725             p.expr0(x.X, reduceDepth(depth)) //
726         } else {
727             p.print(token.LPAREN)
728             p.expr0(x.X, reduceDepth(depth)) //
729             p.print(x.Rparen, token.RPAREN)
730         }
731
732     case *ast.SelectorExpr:

```

```

733         p.expr1(x.X, token.HighestPrec, depth)
734         p.print(token.PERIOD)
735         if line := p.lineFor(x.Sel.Pos()); p.pos.IsV
736             p.print(indent, newline, x.Sel.Pos())
737         } else {
738             p.print(x.Sel.Pos(), x.Sel)
739         }
740
741     case *ast.TypeAssertExpr:
742         p.expr1(x.X, token.HighestPrec, depth)
743         p.print(token.PERIOD, token.LPAREN)
744         if x.Type != nil {
745             p.expr(x.Type)
746         } else {
747             p.print(token.TYPE)
748         }
749         p.print(token.RPAREN)
750
751     case *ast.IndexExpr:
752         // TODO(gri): should treat[] like parentheses
753         p.expr1(x.X, token.HighestPrec, 1)
754         p.print(x.Lbrack, token.LBRACK)
755         p.expr0(x.Index, depth+1)
756         p.print(x.Rbrack, token.RBRACK)
757
758     case *ast.SliceExpr:
759         // TODO(gri): should treat[] like parentheses
760         p.expr1(x.X, token.HighestPrec, 1)
761         p.print(x.Lbrack, token.LBRACK)
762         if x.Low != nil {
763             p.expr0(x.Low, depth+1)
764         }
765         // blanks around ":" if both sides exist and
766         if depth <= 1 && x.Low != nil && x.High != n
767             p.print(blank, token.COLON, blank)
768         } else {
769             p.print(token.COLON)
770         }
771         if x.High != nil {
772             p.expr0(x.High, depth+1)
773         }
774         p.print(x.Rbrack, token.RBRACK)
775
776     case *ast.CallExpr:
777         if len(x.Args) > 1 {
778             depth++
779         }
780         p.expr1(x.Fun, token.HighestPrec, depth)
781         p.print(x.Lparen, token.LPAREN)

```

```

782         if x.Ellipsis.IsValid() {
783             p.exprList(x.Lparen, x.Args, depth,
784                 p.print(x.Ellipsis, token.ELLIPSIS)
785                 if x.Rparen.IsValid() && p.lineFor(x
786                     p.print(token.COMMA, formfee
787                 }
788         } else {
789             p.exprList(x.Lparen, x.Args, depth,
790         }
791         p.print(x.Rparen, token.RPAREN)
792
793     case *ast.CompositeLit:
794         // composite literal elements that are compo
795         if x.Type != nil {
796             p.expr1(x.Type, token.HighestPrec, d
797         }
798         p.print(x.Lbrace, token.LBRACE)
799         p.exprList(x.Lbrace, x.Elts, 1, commaTerm, x
800         // do not insert extra line breaks because c
801         // the closing '}' as it might break the cod
802         // trailing ','
803         p.print(noExtraLinebreak, x.Rbrace, token.RE
804
805     case *ast.Ellipsis:
806         p.print(token.ELLIPSIS)
807         if x.Elt != nil {
808             p.expr(x.Elt)
809         }
810
811     case *ast.ArrayType:
812         p.print(token.LBRACK)
813         if x.Len != nil {
814             p.expr(x.Len)
815         }
816         p.print(token.RBRACK)
817         p.expr(x.Elt)
818
819     case *ast.StructType:
820         p.print(token.STRUCT)
821         p.fieldList(x.Fields, true, x.Incomplete)
822
823     case *ast.FuncType:
824         p.print(token.FUNC)
825         p.signature(x.Params, x.Results)
826
827     case *ast.InterfaceType:
828         p.print(token.INTERFACE)
829         p.fieldList(x.Methods, false, x.Incomplete)
830
831     case *ast.MapType:

```

```

832         p.print(token.MAP, token.LBRACK)
833         p.expr(x.Key)
834         p.print(token.RBRACK)
835         p.expr(x.Value)
836
837     case *ast.ChanType:
838         switch x.Dir {
839             case ast.SEND | ast.RECV:
840                 p.print(token.CHAN)
841             case ast.RECV:
842                 p.print(token.ARROW, token.CHAN)
843             case ast.SEND:
844                 p.print(token.CHAN, token.ARROW)
845         }
846         p.print(blank)
847         p.expr(x.Value)
848
849     default:
850         panic("unreachable")
851 }
852
853 return
854 }
855
856 func (p *printer) expr0(x ast.Expr, depth int) {
857     p.expr1(x, token.LowestPrec, depth)
858 }
859
860 func (p *printer) expr(x ast.Expr) {
861     const depth = 1
862     p.expr1(x, token.LowestPrec, depth)
863 }
864
865 // -----
866 // Statements
867
868 // Print the statement list indented, but without a newline
869 // Extra line breaks between statements in the source are re
870 // empty line is printed between statements.
871 func (p *printer) stmtList(list []ast.Stmt, _indent int, nex
872     // TODO(gri): fix _indent code
873     if _indent > 0 {
874         p.print(indent)
875     }
876     multiLine := false
877     for i, s := range list {
878         // _indent == 0 only for lists of switch/sel
879         // in those cases each clause is a new secti
880         p.linebreak(p.lineFor(s.Pos()), 1, ignore, i

```

```

881         p.stmt(s, nextIsRBrace && i == len(list)-1)
882         multiLine = p.isMultiLine(s)
883     }
884     if _indent > 0 {
885         p.print(unindent)
886     }
887 }
888
889 // block prints an *ast.BlockStmt; it always spans at least
890 func (p *printer) block(s *ast.BlockStmt, indent int) {
891     p.print(s.Pos(), token.LBRACE)
892     p.stmtList(s.List, indent, true)
893     p.linebreak(p.lineFor(s.Rbrace), 1, ignore, true)
894     p.print(s.Rbrace, token.RBRACE)
895 }
896
897 func isTypeName(x ast.Expr) bool {
898     switch t := x.(type) {
899     case *ast.Ident:
900         return true
901     case *ast.SelectorExpr:
902         return isTypeName(t.X)
903     }
904     return false
905 }
906
907 func stripParens(x ast.Expr) ast.Expr {
908     if px, strip := x.(*ast.ParenExpr); strip {
909         // parentheses must not be stripped if there
910         // unparenthesized composite literals starti
911         // a type name
912         ast.Inspect(px.X, func(node ast.Node) bool {
913             switch x := node.(type) {
914             case *ast.ParenExpr:
915                 // parentheses protect enclo
916                 return false
917             case *ast.CompositeLit:
918                 if isTypeName(x.Type) {
919                     strip = false // do
920                 }
921                 return false
922             }
923             // in all other cases, keep inspecti
924             return true
925         })
926     if strip {
927         return stripParens(px.X)
928     }
929 }

```

```

930         return x
931     }
932
933     func (p *printer) controlClause(isForStmt bool, init ast.Str
934         p.print(blank)
935         needsBlank := false
936         if init == nil && post == nil {
937             // no semicolons required
938             if expr != nil {
939                 p.expr(stripParens(expr))
940                 needsBlank = true
941             }
942         } else {
943             // all semicolons required
944             // (they are not separators, print them expl
945             if init != nil {
946                 p.stmt(init, false)
947             }
948             p.print(token.SEMICOLON, blank)
949             if expr != nil {
950                 p.expr(stripParens(expr))
951                 needsBlank = true
952             }
953             if isForStmt {
954                 p.print(token.SEMICOLON, blank)
955                 needsBlank = false
956                 if post != nil {
957                     p.stmt(post, false)
958                     needsBlank = true
959                 }
960             }
961         }
962         if needsBlank {
963             p.print(blank)
964         }
965     }
966
967     // indentList reports whether an expression list would look
968     // were indented wholesale (starting with the very first ele
969     // than starting at the first line break).
970     //
971     func (p *printer) indentList(list []ast.Expr) bool {
972         // Heuristic: indentList returns true if there are m
973         // line element in the list, or if there is any elem
974         // starting on the same line as the previous one end
975         if len(list) >= 2 {
976             var b = p.lineFor(list[0].Pos())
977             var e = p.lineFor(list[len(list)-1].End())
978             if 0 < b && b < e {
979                 // list spans multiple lines

```

```

980         n := 0 // multi-line element count
981         line := b
982         for _, x := range list {
983             xb := p.lineFor(x.Pos())
984             xe := p.lineFor(x.End())
985             if line < xb {
986                 // x is not starting
987                 // line as the previ
988                 return true
989             }
990             if xb < xe {
991                 // x is a multi-line
992                 n++
993             }
994             line = xe
995         }
996         return n > 1
997     }
998 }
999 return false
1000 }
1001
1002 func (p *printer) stmt(stmt ast.Stmt, nextIsRBrace bool) {
1003     p.print(stmt.Pos())
1004
1005     switch s := stmt.(type) {
1006     case *ast.BadStmt:
1007         p.print("BadStmt")
1008
1009     case *ast.DeclStmt:
1010         p.decl(s.Decl)
1011
1012     case *ast.EmptyStmt:
1013         // nothing to do
1014
1015     case *ast.LabeledStmt:
1016         // a "correcting" unindent immediately follo
1017         // is applied before the line break if there
1018         // between (see writeWhitespace)
1019         p.print(unindent)
1020         p.expr(s.Label)
1021         p.print(s.Colon, token.COLON, indent)
1022         if e, isEmpty := s.Stmt.(*ast.EmptyStmt); is
1023             if !nextIsRBrace {
1024                 p.print(newline, e.Pos(), to
1025                 break
1026             }
1027     } else {
1028         p.linebreak(p.lineFor(s.Stmt.Pos()),

```

```

1029         }
1030         p.stmt(s.Stmt, nextIsRBrace)
1031
1032     case *ast.ExprStmt:
1033         const depth = 1
1034         p.expr0(s.X, depth)
1035
1036     case *ast.SendStmt:
1037         const depth = 1
1038         p.expr0(s.Chan, depth)
1039         p.print(blank, s.Arrow, token.ARROW, blank)
1040         p.expr0(s.Value, depth)
1041
1042     case *ast.IncDecStmt:
1043         const depth = 1
1044         p.expr0(s.X, depth+1)
1045         p.print(s.TokPos, s.Tok)
1046
1047     case *ast.AssignStmt:
1048         var depth = 1
1049         if len(s.Lhs) > 1 && len(s.Rhs) > 1 {
1050             depth++
1051         }
1052         p.exprList(s.Pos(), s.Lhs, depth, 0, s.TokPo
1053         p.print(blank, s.TokPos, s.Tok, blank)
1054         p.exprList(s.TokPos, s.Rhs, depth, 0, token.
1055
1056     case *ast.GoStmt:
1057         p.print(token.GO, blank)
1058         p.expr(s.Call)
1059
1060     case *ast.DeferStmt:
1061         p.print(token.DEFER, blank)
1062         p.expr(s.Call)
1063
1064     case *ast.ReturnStmt:
1065         p.print(token.RETURN)
1066         if s.Results != nil {
1067             p.print(blank)
1068             // Use indentList heuristic to make
1069             // better (issue 1207). A more syste
1070             // always indent, but this would cau
1071             // reformatting of the code base and
1072             // lead to more nicely formatted cod
1073             if p.indentList(s.Results) {
1074                 p.print(indent)
1075                 p.exprList(s.Pos(), s.Result
1076                 p.print(unindent)
1077             } else {

```

```

1078                                     p.exprList(s.Pos(), s.Result
1079                                     }
1080                                 }
1081
1082     case *ast.BranchStmt:
1083         p.print(s.Tok)
1084         if s.Label != nil {
1085             p.print(blank)
1086             p.expr(s.Label)
1087         }
1088
1089     case *ast.BlockStmt:
1090         p.block(s, 1)
1091
1092     case *ast.IfStmt:
1093         p.print(token.IF)
1094         p.controlClause(false, s.Init, s.Cond, nil)
1095         p.block(s.Body, 1)
1096         if s.Else != nil {
1097             p.print(blank, token.ELSE, blank)
1098             switch s.Else.(type) {
1099                 case *ast.BlockStmt, *ast.IfStmt:
1100                     p.stmt(s.Else, nextIsRBrace)
1101                 default:
1102                     p.print(token.LBRACE, indent
1103                     p.stmt(s.Else, true)
1104                     p.print(unindent, formfeed,
1105                     }
1106             }
1107
1108     case *ast.CaseClause:
1109         if s.List != nil {
1110             p.print(token.CASE, blank)
1111             p.exprList(s.Pos(), s.List, 1, 0, s.
1112         } else {
1113             p.print(token.DEFAULT)
1114         }
1115         p.print(s.Colon, token.COLON)
1116         p.stmtList(s.Body, 1, nextIsRBrace)
1117
1118     case *ast.SwitchStmt:
1119         p.print(token.SWITCH)
1120         p.controlClause(false, s.Init, s.Tag, nil)
1121         p.block(s.Body, 0)
1122
1123     case *ast.TypeSwitchStmt:
1124         p.print(token.SWITCH)
1125         if s.Init != nil {
1126             p.print(blank)
1127             p.stmt(s.Init, false)

```

```

1128         p.print(token.SEMICOLON)
1129     }
1130     p.print(blank)
1131     p.stmt(s.Assign, false)
1132     p.print(blank)
1133     p.block(s.Body, 0)
1134
1135     case *ast.CommClause:
1136         if s.Comm != nil {
1137             p.print(token.CASE, blank)
1138             p.stmt(s.Comm, false)
1139         } else {
1140             p.print(token.DEFAULT)
1141         }
1142         p.print(s.Colon, token.COLON)
1143         p.stmtList(s.Body, 1, nextIsRBrace)
1144
1145     case *ast.SelectStmt:
1146         p.print(token.SELECT, blank)
1147         body := s.Body
1148         if len(body.List) == 0 && !p.commentBefore(p
1149             // print empty select statement w/o
1150             p.print(body.Lbrace, token.LBRACE, b
1151         } else {
1152             p.block(body, 0)
1153         }
1154
1155     case *ast.ForStmt:
1156         p.print(token.FOR)
1157         p.controlClause(true, s.Init, s.Cond, s.Post
1158         p.block(s.Body, 1)
1159
1160     case *ast.RangeStmt:
1161         p.print(token.FOR, blank)
1162         p.expr(s.Key)
1163         if s.Value != nil {
1164             // use position of value following t
1165             // comma position for correct commen
1166             p.print(s.Value.Pos(), token.COMMA,
1167             p.expr(s.Value)
1168         }
1169         p.print(blank, s.TokPos, s.Tok, blank, token
1170         p.expr(stripParens(s.X))
1171         p.print(blank)
1172         p.block(s.Body, 1)
1173
1174     default:
1175         panic("unreachable")
1176 }

```

```

1177
1178         return
1179     }
1180
1181 // -----
1182 // Declarations
1183
1184 // The keepTypeColumn function determines if the type column
1185 // consecutive const or var declarations must be kept, or if
1186 // values (V) can be placed in the type column (T) instead.
1187 // in the result slice is true if the type column in spec[i]
1188 //
1189 // For example, the declaration:
1190 //
1191 //     const (
1192 //         foobar int = 42 // comment
1193 //         x         = 7  // comment
1194 //         foo
1195 //         bar = 991
1196 //     )
1197 //
1198 // leads to the type/values matrix below. A run of value col
1199 // be moved into the type column if there is no type for any
1200 // in that column (we only move entire columns so that they
1201 //
1202 //     matrix           formatted           result
1203 //     T V   ->   T V   ->
1204 //     - V           - V           true     there is a T a
1205 //     - -           - -           true     column must be
1206 //     - V           V -           false
1207 //                                     false   V is moved int
1208 //
1209 func keepTypeColumn(specs []ast.Spec) []bool {
1210     m := make([]bool, len(specs))
1211
1212     populate := func(i, j int, keepType bool) {
1213         if keepType {
1214             for ; i < j; i++ {
1215                 m[i] = true
1216             }
1217         }
1218     }
1219
1220     i0 := -1 // if i0 >= 0 we are in a run and i0 is the
1221     var keepType bool
1222     for i, s := range specs {
1223         t := s.(*ast.ValueSpec)
1224         if t.Values != nil {
1225             if i0 < 0 {

```

```

1226             // start of a run of ValueSp
1227             i0 = i
1228             keepType = false
1229         }
1230     } else {
1231         if i0 >= 0 {
1232             // end of a run
1233             populate(i0, i, keepType)
1234             i0 = -1
1235         }
1236     }
1237     if t.Type != nil {
1238         keepType = true
1239     }
1240 }
1241 if i0 >= 0 {
1242     // end of a run
1243     populate(i0, len(specs), keepType)
1244 }
1245
1246 return m
1247 }
1248
1249 func (p *printer) valueSpec(s *ast.ValueSpec, keepType bool)
1250     p.setComment(s.Doc)
1251     p.identList(s.Names, false) // always present
1252     extraTabs := 3
1253     if s.Type != nil || keepType {
1254         p.print(vtab)
1255         extraTabs--
1256     }
1257     if s.Type != nil {
1258         p.expr(s.Type)
1259     }
1260     if s.Values != nil {
1261         p.print(vtab, token.ASSIGN, blank)
1262         p.exprList(token.NoPos, s.Values, 1, 0, token.NoPos)
1263         extraTabs--
1264     }
1265     if s.Comment != nil {
1266         for ; extraTabs > 0; extraTabs-- {
1267             p.print(vtab)
1268         }
1269         p.setComment(s.Comment)
1270     }
1271 }
1272
1273 // The parameter n is the number of specs in the group. If d
1274 // multi-line identifier lists in the spec are indented when
1275 // linebreak is encountered.

```

```

1276 //
1277 func (p *printer) spec(spec ast.Spec, n int, doIndent bool)
1278     switch s := spec.(type) {
1279     case *ast.ImportSpec:
1280         p.setComment(s.Doc)
1281         if s.Name != nil {
1282             p.expr(s.Name)
1283             p.print(blank)
1284         }
1285         p.expr(s.Path)
1286         p.setComment(s.Comment)
1287         p.print(s.EndPos)
1288
1289     case *ast.ValueSpec:
1290         if n != 1 {
1291             p.internalError("expected n = 1; got
1292
1293         }
1294         p.setComment(s.Doc)
1295         p.identList(s.Names, doIndent) // always pre
1296         if s.Type != nil {
1297             p.print(blank)
1298             p.expr(s.Type)
1299         }
1300         if s.Values != nil {
1301             p.print(blank, token.ASSIGN, blank)
1302             p.exprList(token.NoPos, s.Values, 1,
1303
1304         }
1305         p.setComment(s.Comment)
1306
1307     case *ast.TypeSpec:
1308         p.setComment(s.Doc)
1309         p.expr(s.Name)
1310         if n == 1 {
1311             p.print(blank)
1312         } else {
1313             p.print(vtab)
1314         }
1315         p.expr(s.Type)
1316         p.setComment(s.Comment)
1317
1318     default:
1319         panic("unreachable")
1320 }
1321 }
1322
1323 func (p *printer) genDecl(d *ast.GenDecl) {
1324     p.setComment(d.Doc)
1325     p.print(d.Pos(), d.Tok, blank)

```

```

1325     if d.Lparen.IsValid() {
1326         // group of parenthesized declarations
1327         p.print(d.Lparen, token.LPAREN)
1328         if n := len(d.Specs); n > 0 {
1329             p.print(indent, formfeed)
1330             if n > 1 && (d.Tok == token.CONST ||
1331                 // two or more grouped const
1332                 // determine if the type col
1333                 keepType := keepTypeColumn(d
1334                 newSection := false
1335                 for i, s := range d.Specs {
1336                     if i > 0 {
1337                         p.linebreak(
1338                             }
1339                             p.valueSpec(s.(*ast.
1340                             newSection = p.isMul
1341                         }
1342                     } else {
1343                         newSection := false
1344                         for i, s := range d.Specs {
1345                             if i > 0 {
1346                                 p.linebreak(
1347                                     }
1348                                     p.spec(s, n, false)
1349                                     newSection = p.isMul
1350                                 }
1351                             }
1352                         p.print(unindent, formfeed)
1353                     }
1354                     p.print(d.Rparen, token.RPAREN)
1355                 } else {
1356                     // single declaration
1357                     p.spec(d.Specs[0], 1, true)
1358                 }
1359             }
1360         }
1361
1362 // nodeSize determines the size of n in chars after formatti
1363 // The result is <= maxSize if the node fits on one line wit
1364 // most maxSize chars and the formatted output doesn't conta
1365 // any control chars. Otherwise, the result is > maxSize.
1366 //
1367 func (p *printer) nodeSize(n ast.Node, maxSize int) (size in
1368 // nodeSize invokes the printer, which may invoke no
1369 // recursively. For deep composite literal nests, th
1370 // lead to an exponential algorithm. Remember previo
1371 // results to prune the recursion (was issue 1628).
1372 if size, found := p.nodeSizes[n]; found {
1373     return size

```

```

1374     }
1375
1376     size = maxSize + 1 // assume n doesn't fit
1377     p.nodeSizes[n] = size
1378
1379     // nodeSize computation must be independent of parti
1380     // style so that we always get the same decision; pr
1381     // in RawFormat
1382     cfg := Config{Mode: RawFormat}
1383     var buf bytes.Buffer
1384     if err := cfg.fprint(&buf, p.fset, n, p.nodeSizes);
1385         return
1386     }
1387     if buf.Len() <= maxSize {
1388         for _, ch := range buf.Bytes() {
1389             if ch < ' ' {
1390                 return
1391             }
1392         }
1393         size = buf.Len() // n fits
1394         p.nodeSizes[n] = size
1395     }
1396     return
1397 }
1398
1399 func (p *printer) isOneLineFunc(b *ast.BlockStmt, headerSize
1400     pos1 := b.Pos()
1401     pos2 := b.Rbrace
1402     if pos1.IsValid() && pos2.IsValid() && p.lineFor(pos
1403         // opening and closing brace are on differen
1404         return false
1405     }
1406     if len(b.List) > 5 || p.commentBefore(p.posFor(pos2)
1407         // too many statements or there is a comment
1408         return false
1409     }
1410     // otherwise, estimate body size
1411     const maxSize = 100
1412     bodySize := 0
1413     for i, s := range b.List {
1414         if i > 0 {
1415             bodySize += 2 // space for a semicol
1416         }
1417         bodySize += p.nodeSize(s, maxSize)
1418     }
1419     return headerSize+bodySize <= maxSize
1420 }
1421
1422 func (p *printer) funcBody(b *ast.BlockStmt, headerSize int,
1423     if b == nil {

```

```

1424         return
1425     }
1426
1427     if p.isOneLineFunc(b, headerSize) {
1428         sep := vtab
1429         if isLit {
1430             sep = blank
1431         }
1432         p.print(sep, b.Lbrace, token.LBRACE)
1433         if len(b.List) > 0 {
1434             p.print(blank)
1435             for i, s := range b.List {
1436                 if i > 0 {
1437                     p.print(token.SEMIC)
1438                 }
1439                 p.stmt(s, i == len(b.List)-1)
1440             }
1441             p.print(blank)
1442         }
1443         p.print(b.Rbrace, token.RBRACE)
1444         return
1445     }
1446
1447     p.print(blank)
1448     p.block(b, 1)
1449 }
1450
1451 // distance returns the column difference between from and t
1452 // are on the same line; if they are on different lines (or
1453 // the result is infinity.
1454 func (p *printer) distance(from token.Pos, to token.Position) int {
1455     from := p.posFor(from)
1456     if from.IsValid() && to.IsValid() && from.Line == to.Line {
1457         return to.Column - from.Column
1458     }
1459     return infinity
1460 }
1461
1462 func (p *printer) funcDecl(d *ast.FuncDecl) {
1463     p.setComment(d.Doc)
1464     p.print(d.Pos(), token.FUNC, blank)
1465     if d.Recv != nil {
1466         p.parameters(d.Recv) // method: print receiver
1467         p.print(blank)
1468     }
1469     p.expr(d.Name)
1470     p.signature(d.Type.Params, d.Type.Results)
1471     p.funcBody(d.Body, p.distance(d.Pos(), p.pos), false)
1472 }

```

```

1473
1474 func (p *printer) decl(decl ast.Decl) {
1475     switch d := decl.(type) {
1476     case *ast.BadDecl:
1477         p.print(d.Pos(), "BadDecl")
1478     case *ast.GenDecl:
1479         p.genDecl(d)
1480     case *ast.FuncDecl:
1481         p.funcDecl(d)
1482     default:
1483         panic("unreachable")
1484     }
1485 }
1486
1487 // -----
1488 // Files
1489
1490 func declToken(decl ast.Decl) (tok token.Token) {
1491     tok = token.ILLEGAL
1492     switch d := decl.(type) {
1493     case *ast.GenDecl:
1494         tok = d.Tok
1495     case *ast.FuncDecl:
1496         tok = token.FUNC
1497     }
1498     return
1499 }
1500
1501 func (p *printer) file(src *ast.File) {
1502     p.setComment(src.Doc)
1503     p.print(src.Pos(), token.PACKAGE, blank)
1504     p.expr(src.Name)
1505
1506     if len(src.Decls) > 0 {
1507         tok := token.ILLEGAL
1508         for _, d := range src.Decls {
1509             prev := tok
1510             tok = declToken(d)
1511             // if the declaration token changed
1512             // or the next declaration has docum
1513             // print an empty line between top-l
1514             // (because p.linebreak is called wi
1515             // is past any documentation, the mi
1516             // even w/o the extra getDoc(d) nil-
1517             // linebreak logic improves - there'
1518             min := 1
1519             if prev != tok || getDoc(d) != nil {
1520                 min = 2
1521             }

```

```
1522             p.linebreak(p.lineFor(d.Pos()), min,  
1523             p.decl(d)  
1524         }  
1525     }  
1526  
1527     p.print(newline)  
1528 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/go/printer/printer.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package printer implements printing of AST nodes.
6 package printer
7
8 import (
9     "fmt"
10    "go/ast"
11    "go/token"
12    "io"
13    "os"
14    "strconv"
15    "strings"
16    "text/tabwriter"
17 )
18
19 const (
20     maxNewlines = 2 // max. number of newlines between
21     debug       = false // enable for debugging
22     infinity    = 1 << 30
23 )
24
25 type whiteSpace byte
26
27 const (
28     ignore    = whiteSpace(0)
29     blank     = whiteSpace(' ')
30     vtab      = whiteSpace('\v')
31     newline   = whiteSpace('\n')
32     formfeed  = whiteSpace('\f')
33     indent    = whiteSpace('>')
34     unindent  = whiteSpace('<')
35 )
36
37 // A pmode value represents the current printer mode.
38 type pmode int
39
40 const (
41     noExtraLinebreak pmode = 1 << iota
```

```

42 )
43
44 type printer struct {
45     // Configuration (does not change after initializati
46     Config
47     fset *token.FileSet
48
49     // Current state
50     output []byte // raw printer result
51     indent int // current indentation
52     mode pmode // current printer mode
53     impliedSemi bool // if set, a linebreak impl
54     lastTok token.Token // the last token printed (
55     wsbuf []whiteSpace // delayed white space
56
57     // Positions
58     // The out position differs from the pos position wh
59     // formatting differs from the source formatting (in
60     // white space). If there's a difference and SourceP
61     // ConfigMode, //line comments are used in the outpu
62     // original source positions for a reader.
63     pos token.Position // current position in AST (sour
64     out token.Position // current position in output sp
65     last token.Position // value of pos after calling wr
66
67     // The list of all source comments, in order of appe
68     comments []*ast.CommentGroup // may be nil
69     cindex int // current comme
70     useNodeComments bool // if not set, i
71
72     // Information about p.comments[p.cindex]; set up by
73     comment *ast.CommentGroup // = p.comments[p.c
74     commentOffset int // = p.posFor(p.comr
75     commentNewline bool // true if the comr
76
77     // Cache of already computed node sizes.
78     nodeSizes map[ast.Node]int
79
80     // Cache of most recently computed line position.
81     cachedPos token.Pos
82     cachedLine int // line corresponding to cachedPos
83 }
84
85 func (p *printer) init(cfg *Config, fset *token.FileSet, nod
86     p.Config = *cfg
87     p.fset = fset
88     p.pos = token.Position{Line: 1, Column: 1}
89     p.out = token.Position{Line: 1, Column: 1}
90     p.wsbuf = make([]whiteSpace, 0, 16) // whitespace se
91     p.nodeSizes = nodeSizes

```

```

92         p.cachedPos = -1
93     }
94
95     // commentsHaveNewline reports whether a list of comments be
96     // an *ast.CommentGroup contains newlines. Because the posit
97     // may only be partially correct, we also have to read the c
98     func (p *printer) commentsHaveNewline(list []*ast.Comment) b
99         // len(list) > 0
100        line := p.lineFor(list[0].Pos())
101        for i, c := range list {
102            if i > 0 && p.lineFor(list[i].Pos()) != line
103                // not all comments on the same line
104                return true
105        }
106        if t := c.Text; len(t) >= 2 && (t[1] == '/')
107            return true
108        }
109    }
110    _ = line
111    return false
112 }
113
114 func (p *printer) nextComment() {
115     for p.cindex < len(p.comments) {
116         c := p.comments[p.cindex]
117         p.cindex++
118         if list := c.List; len(list) > 0 {
119             p.comment = c
120             p.commentOffset = p.posFor(list[0].P
121             p.commentNewline = p.commentsHaveNew
122             return
123         }
124         // we should not reach here (correct ASTs do
125         // ast.CommentGroup nodes), but be conservat
126     }
127     // no more comments
128     p.commentOffset = infinity
129 }
130
131 func (p *printer) internalError(msg ...interface{}) {
132     if debug {
133         fmt.Print(p.pos.String() + ": ")
134         fmt.Println(msg...)
135         panic("go/printer")
136     }
137 }
138
139 func (p *printer) posFor(pos token.Pos) token.Position {
140     // not used frequently enough to cache entire token.

```

```

141         return p.fset.Position(pos)
142     }
143
144     func (p *printer) lineFor(pos token.Pos) int {
145         if pos != p.cachedPos {
146             p.cachedPos = pos
147             p.cachedLine = p.fset.Position(pos).Line
148         }
149         return p.cachedLine
150     }
151
152     // atLineBegin emits a //line comment if necessary and print
153     func (p *printer) atLineBegin(pos token.Position) {
154         // write a //line comment if necessary
155         if p.Config.Mode&SourcePos != 0 && pos.IsValid() &&
156             p.output = append(p.output, tabwriter.Escape
157             p.output = append(p.output, fmt.Sprintf("//l
158             p.output = append(p.output, tabwriter.Escape
159             // p.out must match the //line comment
160             p.out.Filename = pos.Filename
161             p.out.Line = pos.Line
162         }
163
164         // write indentation
165         // use "hard" htabs - indentation columns
166         // must not be discarded by the tabwriter
167         for i := 0; i < p.indent; i++ {
168             p.output = append(p.output, '\t')
169         }
170
171         // update positions
172         i := p.indent
173         p.pos.Offset += i
174         p.pos.Column += i
175         p.out.Column += i
176     }
177
178     // writeByte writes ch n times to p.output and updates p.pos
179     func (p *printer) writeByte(ch byte, n int) {
180         if p.out.Column == 1 {
181             p.atLineBegin(p.pos)
182         }
183
184         for i := 0; i < n; i++ {
185             p.output = append(p.output, ch)
186         }
187
188         // update positions
189         p.pos.Offset += n

```

```

190         if ch == '\n' || ch == '\f' {
191             p.pos.Line += n
192             p.out.Line += n
193             p.pos.Column = 1
194             p.out.Column = 1
195             return
196         }
197         p.pos.Column += n
198         p.out.Column += n
199     }
200
201     // writeString writes the string s to p.output and updates p
202     // and p.last. If isLit is set, s is escaped w/ tabwriter.Es
203     // to protect s from being interpreted by the tabwriter.
204     //
205     // Note: writeString is only used to write Go tokens, litera
206     // comments, all of which must be written literally. Thus, i
207     // to always set isLit = true. However, setting it explicitl
208     // needed (i.e., when we don't know that s contains no tabs
209     // avoids processing extra escape characters and reduces run
210     // printer benchmark by up to 10%.
211     //
212     func (p *printer) writeString(pos token.Position, s string,
213         if p.out.Column == 1 {
214             p.atLineBegin(pos)
215         }
216
217         if pos.IsValid() {
218             // update p.pos (if pos is invalid, continue
219             // Note: Must do this after handling line be
220             // atLineBegin updates p.pos if there's inde
221             // is the position of s.
222             p.pos = pos
223             // reset state if the file changed
224             // (used when printing merged ASTs of differ
225             // e.g., the result of ast.MergePackageFiles
226             if p.last.IsValid() && p.last.Filename != pc
227                 p.indent = 0
228                 p.mode = 0
229                 p.wsbuf = p.wsbuf[0:0]
230         }
231     }
232
233     if isLit {
234         // Protect s such that is passes through the
235         // unchanged. Note that valid Go programs ca
236         // tabwriter.Escape bytes since they do not
237         // UTF-8 sequences.
238         p.output = append(p.output, tabwriter.Escape
239     }

```

```

240
241     if debug {
242         p.output = append(p.output, fmt.Sprintf("/**%
243     }
244     p.output = append(p.output, s...)
245
246     // update positions
247     nlines := 0
248     var li int // index of last newline; valid if nlines
249     for i := 0; i < len(s); i++ {
250         // Go tokens cannot contain '\f' - no need t
251         if s[i] == '\n' {
252             nlines++
253             li = i
254         }
255     }
256     p.pos.Offset += len(s)
257     if nlines > 0 {
258         p.pos.Line += nlines
259         p.out.Line += nlines
260         c := len(s) - li
261         p.pos.Column = c
262         p.out.Column = c
263     } else {
264         p.pos.Column += len(s)
265         p.out.Column += len(s)
266     }
267
268     if isLit {
269         p.output = append(p.output, tabwriter.Escape
270     }
271
272     p.last = p.pos
273 }
274
275 // writeCommentPrefix writes the whitespace before a comment
276 // If there is any pending whitespace, it consumes as much o
277 // it as is likely to help position the comment nicely.
278 // pos is the comment position, next the position of the ite
279 // after all pending comments, prev is the previous comment
280 // a group of comments (or nil), and tok is the next token.
281 //
282 func (p *printer) writeCommentPrefix(pos, next token.Positic
283     if len(p.output) == 0 {
284         // the comment is the first item to be print
285         return
286     }
287
288     if pos.IsValid() && pos.Filename != p.last.Filename

```

```

289         // comment in a different file - separate wi
290         p.WriteByte('\f', maxNewlines)
291         return
292     }
293
294     if pos.Line == p.last.Line && (prev == nil || prev.T
295         // comment on the same line as last item:
296         // separate with at least one separator
297         hasSep := false
298         if prev == nil {
299             // first comment of a comment group
300             j := 0
301             for i, ch := range p.wsbuf {
302                 switch ch {
303                     case blank:
304                         // ignore any blanks
305                         p.wsbuf[i] = ignore
306                         continue
307                     case vtab:
308                         // respect existing
309                         // for proper format
310                         hasSep = true
311                         continue
312                     case indent:
313                         // apply pending ind
314                         continue
315                 }
316                 j = i
317                 break
318             }
319             p.writeWhitespace(j)
320         }
321         // make sure there is at least one separator
322         if !hasSep {
323             sep := byte('\t')
324             if pos.Line == next.Line {
325                 // next item is on the same
326                 // (which must be a /*-style
327                 // with a blank instead of a
328                 sep = ' '
329             }
330             p.WriteByte(sep, 1)
331         }
332     } else {
333         // comment on a different line:
334         // separate with at least one line break
335         droppedLinebreak := false
336         j := 0

```

```

338     for i, ch := range p.wsbuf {
339         switch ch {
340             case blank, vtab:
341                 // ignore any horizontal whi
342                 p.wsbuf[i] = ignore
343                 continue
344             case indent:
345                 // apply pending indentation
346                 continue
347             case unindent:
348                 // if this is not the last u
349                 // as it is (likely) belongi
350                 // construct (e.g., a multi-
351                 // and is not part of closin
352                 if i+1 < len(p.wsbuf) && p.w
353                     continue
354             }
355             // if the next token is not
356             // if it appears that the co
357             // token; otherwise assume t
358             // closing block and stop (t
359             // comments before a case la
360             // apply to the next case in
361             if tok != token.RBRACE && pc
362                 continue
363         }
364         case newline, formfeed:
365             p.wsbuf[i] = ignore
366             droppedLinebreak = prev == n
367         }
368         j = i
369         break
370     }
371     p.writeWhitespace(j)
372
373     // determine number of linebreaks before the
374     n := 0
375     if pos.IsValid() && p.last.IsValid() {
376         n = pos.Line - p.last.Line
377         if n < 0 { // should never happen
378             n = 0
379         }
380     }
381
382     // at the package scope level only (p.indent
383     // add an extra newline if we dropped one be
384     // this preserves a blank line before docume
385     // comments at the package scope level (issu
386     if p.indent == 0 && droppedLinebreak {
387         n++

```

```

388         }
389
390         // make sure there is at least one line brea
391         // if the previous comment was a line commen
392         if n == 0 && prev != nil && prev.Text[1] ==
393             n = 1
394     }
395
396     if n > 0 {
397         // use formfeeds to break columns be
398         // this is analogous to using formfe
399         // individual lines of /*-style commr
400         p.WriteByte('\f', nlimit(n))
401     }
402 }
403 }
404
405 // Split comment text into lines
406 // (using strings.Split(text, "\n") is significantly slower
407 // this specific purpose, as measured with: go test -bench=P
408 func split(text string) []string {
409     // count lines (comment text never ends in a newline
410     n := 1
411     for i := 0; i < len(text); i++ {
412         if text[i] == '\n' {
413             n++
414         }
415     }
416
417     // split
418     lines := make([]string, n)
419     n = 0
420     i := 0
421     for j := 0; j < len(text); j++ {
422         if text[j] == '\n' {
423             lines[n] = text[i:j] // exclude newl
424             i = j + 1           // discard newl
425             n++
426         }
427     }
428     lines[n] = text[i:]
429
430     return lines
431 }
432
433 // Returns true if s contains only white space
434 // (only tabs and blanks can appear in the printer's context
435 func isBlank(s string) bool {
436     for i := 0; i < len(s); i++ {

```

```

437         if s[i] > ' ' {
438             return false
439         }
440     }
441     return true
442 }
443
444 func commonPrefix(a, b string) string {
445     i := 0
446     for i < len(a) && i < len(b) && a[i] == b[i] && (a[i]
447         i++)
448     }
449     return a[0:i]
450 }
451
452 func stripCommonPrefix(lines []string) {
453     if len(lines) < 2 {
454         return // at most one line - nothing to do
455     }
456     // len(lines) >= 2
457
458     // The heuristic in this function tries to handle a
459     // common patterns of /*-style comments: Comments wh
460     // the opening /* and closing */ are aligned and the
461     // rest of the comment text is aligned and indented
462     // blanks or tabs, cases with a vertical "line of st
463     // on the left, and cases where the closing */ is on
464     // same line as the last comment text.
465
466     // Compute maximum common white prefix of all but th
467     // last, and blank lines, and replace blank lines wi
468     // lines (the first line starts with /* and has no p
469     // In case of two-line comments, consider the last l
470     // the prefix computation since otherwise the prefix
471     // be empty.
472     //
473     // Note that the first and last line are never empty
474     // contain the opening /* and closing */ respectivel
475     // thus they can be ignored by the blank line check.
476     var prefix string
477     if len(lines) > 2 {
478         first := true
479         for i, line := range lines[1 : len(lines)-1]
480             switch {
481             case isBlank(line):
482                 lines[1+i] = "" // range sta
483             case first:
484                 prefix = commonPrefix(line,
485                     first = false

```

```

486             default:
487                 prefix = commonPrefix(prefix
488             )
489         }
490     } else { // len(lines) == 2, lines cannot be blank (
491         line := lines[1]
492         prefix = commonPrefix(line, line)
493     }
494
495     /*
496     * Check for vertical "line of stars" and correct pr
497     */
498     lineOfStars := false
499     if i := strings.Index(prefix, "*"); i >= 0 {
500         // Line of stars present.
501         if i > 0 && prefix[i-1] == ' ' {
502             i-- // remove trailing blank from pr
503         }
504         prefix = prefix[0:i]
505         lineOfStars = true
506     } else {
507         // No line of stars present.
508         // Determine the white space on the first li
509         // and before the beginning of the comment t
510         // blanks instead of the /* unless the first
511         // the /* is a tab. If the first comment lin
512         // for the opening /*, assume up to 3 blanks
513         // whitespace may be found as suffix in the
514         first := lines[0]
515         if isBlank(first[2:]) {
516             // no comment text on the first line
517             // reduce prefix by up to 3 blanks o
518             // if present - this keeps comment t
519             // relative to the /* and */'s if it
520             // in the first place
521             i := len(prefix)
522             for n := 0; n < 3 && i > 0 && prefix
523                 i--
524             }
525             if i == len(prefix) && i > 0 && pref
526                 i--
527             }
528             prefix = prefix[0:i]
529         } else {
530             // comment text on the first line
531             suffix := make([]byte, len(first))
532             n := 2 // start after opening /*
533             for n < len(first) && first[n] <= '
534                 suffix[n] = first[n]
535                 n++

```

```

536         }
537         if n > 2 && suffix[2] == '\t' {
538             // assume the '\t' compensat
539             suffix = suffix[2:n]
540         } else {
541             // otherwise assume two blan
542             suffix[0], suffix[1] = ' ',
543             suffix = suffix[0:n]
544         }
545         // Shorten the computed common prefi
546         // suffix, if it is found as suffix
547         if strings.HasSuffix(prefix, string(
548             prefix = prefix[0 : len(pref
549         )
550     }
551 }
552
553 // Handle last line: If it only contains a closing *
554 // with the opening /*, otherwise align the text wit
555 // lines.
556 last := lines[len(lines)-1]
557 closing := "*/"
558 i := strings.Index(last, closing) // i >= 0 (closing
559 if isBlank(last[0:i]) {
560     // last line only contains closing */
561     if lineOfStars {
562         closing = " */" // add blank to align
563     }
564     lines[len(lines)-1] = prefix + closing
565 } else {
566     // last line contains more comment text - as
567     // it is aligned like the other lines and in
568     // in prefix computation
569     prefix = commonPrefix(prefix, last)
570 }
571
572 // Remove the common prefix from all but the first a
573 for i, line := range lines[1:] {
574     if len(line) != 0 {
575         lines[1+i] = line[len(prefix):] // r
576     }
577 }
578 }
579
580 func (p *printer) writeComment(comment *ast.Comment) {
581     text := comment.Text
582     pos := p.posFor(comment.Pos())
583
584     const linePrefix = "//line "

```

```

585     if strings.HasPrefix(text, linePrefix) && (!pos.IsVa
586         // possibly a line directive
587         ldir := strings.TrimSpace(text[len(linePrefi
588         if i := strings.LastIndex(ldir, ":"); i >= 0
589             if line, err := strconv.Atoi(ldir[i+
590                 // The line directive we are
591                 // the Filename and Line num
592                 // tokens. We have to update
593                 // accordingly and suspend i
594                 indent := p.indent
595                 p.indent = 0
596                 defer func() {
597                     p.pos.Filename = ldi
598                     p.pos.Line = line
599                     p.pos.Column = 1
600                     p.indent = indent
601                 }()
602             }
603         }
604     }
605
606     // shortcut common case of //-style comments
607     if text[1] == '/' {
608         p.writeString(pos, text, true)
609         return
610     }
611
612     // for /*-style comments, print line by line and let
613     // write function take care of the proper indentatio
614     lines := split(text)
615     stripCommonPrefix(lines)
616
617     // write comment lines, separated by formfeed,
618     // without a line break after the last line
619     for i, line := range lines {
620         if i > 0 {
621             p.WriteByte('\f', 1)
622             pos = p.pos
623         }
624         if len(line) > 0 {
625             p.writeString(pos, line, true)
626         }
627     }
628 }
629
630 // writeCommentSuffix writes a line break after a comment if
631 // and processes any leftover indentation information. If a
632 // is needed, the kind of break (newline vs formfeed) depend
633 // pending whitespace. The writeCommentSuffix result indicat

```

```

634 // newline was written or if a formfeed was dropped from the
635 // buffer.
636 //
637 func (p *printer) writeCommentSuffix(needsLinebreak bool) (w
638     for i, ch := range p.wsbuf {
639         switch ch {
640             case blank, vtab:
641                 // ignore trailing whitespace
642                 p.wsbuf[i] = ignore
643             case indent, unindent:
644                 // don't lose indentation informatio
645             case newline, formfeed:
646                 // if we need a line break, keep exa
647                 // but remember if we dropped any fo
648                 if needsLinebreak {
649                     needsLinebreak = false
650                     wroteNewline = true
651                 } else {
652                     if ch == formfeed {
653                         droppedFF = true
654                     }
655                     p.wsbuf[i] = ignore
656                 }
657             }
658         }
659     p.writeWhitespace(len(p.wsbuf))
660
661     // make sure we have a line break
662     if needsLinebreak {
663         p.WriteByte('\n', 1)
664         wroteNewline = true
665     }
666
667     return
668 }
669
670 // intersperseComments consumes all comments that appear bef
671 // tok and prints it together with the buffered whitespace (
672 // that needs to be written before the next token). A heuris
673 // the comments and whitespace. The intersperseComments resu
674 // newline was written or if a formfeed was dropped from the
675 //
676 func (p *printer) intersperseComments(next token.Position, t
677     var last *ast.Comment
678     for p.commentBefore(next) {
679         for _, c := range p.comment.List {
680             p.writeCommentPrefix(p.posFor(c.Pos(
681                 p.writeComment(c)
682                 last = c
683         }

```

```

684         p.nextComment()
685     }
686
687     if last != nil {
688         // if the last comment is a /*-style comment
689         // follows on the same line but is not a com
690         // token, add an extra blank for separation
691         if last.Text[1] == '*' && p.lineFor(last.Pos
692             tok != token.RPAREN && tok != token.
693             p.WriteByte(' ', 1)
694     }
695     // ensure that there is a line break after a
696     // before a closing '}' unless explicitly di
697     needsLinebreak :=
698         last.Text[1] == '/' ||
699         tok == token.RBRACE && p.mod
700         tok == token.EOF
701     return p.writeCommentSuffix(needsLinebreak)
702 }
703
704 // no comment was written - we should never reach he
705 // intersperseComments should not be called in that
706 p.internalError("intersperseComments called without
707 return
708 }
709
710 // whiteWhitespace writes the first n whitespace entries.
711 func (p *printer) writeWhitespace(n int) {
712     // write entries
713     for i := 0; i < n; i++ {
714         switch ch := p.wsbuf[i]; ch {
715             case ignore:
716                 // ignore!
717             case indent:
718                 p.indent++
719             case unindent:
720                 p.indent--
721                 if p.indent < 0 {
722                     p.internalError("negative in
723                     p.indent = 0
724                 }
725             case newline, formfeed:
726                 // A line break immediately followed
727                 // unindent is swapped with the unin
728                 // proper label positioning. If a co
729                 // the line break and the label, the
730                 // part of the comment whitespace pr
731                 // will be positioned correctly inde
732                 if i+1 < n && p.wsbuf[i+1] == unde

```

```

733             // Use a formfeed to termina
734             // Otherwise, a long label n
735             // to a wide column may incr
736             // of lines before the label
737             // indentation.
738             p.wsbuf[i], p.wsbuf[i+1] = u
739             i-- // do it again
740             continue
741         }
742         fallthrough
743         default:
744             p.WriteByte(byte(ch), 1)
745     }
746 }
747
748 // shift remaining entries down
749 i := 0
750 for ; n < len(p.wsbuf); n++ {
751     p.wsbuf[i] = p.wsbuf[n]
752     i++
753 }
754 p.wsbuf = p.wsbuf[0:i]
755 }
756
757 // -----
758 // Printing interface
759
760 // nlines limits n to maxNewlines.
761 func nlimit(n int) int {
762     if n > maxNewlines {
763         n = maxNewlines
764     }
765     return n
766 }
767
768 func mayCombine(prev token.Token, next byte) (b bool) {
769     switch prev {
770     case token.INT:
771         b = next == '.' // 1.
772     case token.ADD:
773         b = next == '+' // ++
774     case token.SUB:
775         b = next == '-' // --
776     case token.QUO:
777         b = next == '*' // /*
778     case token.LSS:
779         b = next == '-' || next == '<' // <- or <<
780     case token.AND:
781         b = next == '&' || next == '^' // && or &^

```

```

782     }
783     return
784 }
785
786 // print prints a list of "items" (roughly corresponding to
787 // tokens, but also including whitespace and formatting info
788 // It is the only print function that should be called direc
789 // any of the AST printing functions in nodes.go.
790 //
791 // Whitespace is accumulated until a non-whitespace token ap
792 // comments that need to appear before that token are printe
793 // taking into account the amount and structure of any pendi
794 // space for best comment placement. Then, any leftover whit
795 // printed, followed by the actual token.
796 //
797 func (p *printer) print(args ...interface{}) {
798     for _, arg := range args {
799         // information about the current arg
800         var data string
801         var isLit bool
802         var impliedSemi bool // value for p.impliedS
803
804         switch x := arg.(type) {
805         case pmode:
806             // toggle printer mode
807             p.mode ^= x
808             continue
809
810         case whiteSpace:
811             if x == ignore {
812                 // don't add ignore's to the
813                 // may screw up "correcting"
814                 // LabeledStmt)
815                 continue
816             }
817             i := len(p.wsbuf)
818             if i == cap(p.wsbuf) {
819                 // Whitespace sequences are
820                 // never happen. Handle grac
821                 // bad comment placement) if
822                 p.writeWhitespace(i)
823                 i = 0
824             }
825             p.wsbuf = p.wsbuf[0 : i+1]
826             p.wsbuf[i] = x
827             if x == newline || x == formfeed {
828                 // newlines affect the curre
829                 // and not the state after p
830                 // because comments can be i
831                 // in this case

```

```

832         p.implicitSemi = false
833     }
834     p.lastTok = token.ILLEGAL
835     continue
836
837     case *ast.Ident:
838         data = x.Name
839         implicitSemi = true
840         p.lastTok = token.IDENT
841
842     case *ast.BasicLit:
843         data = x.Value
844         isLit = true
845         implicitSemi = true
846         p.lastTok = x.Kind
847
848     case token.Token:
849         s := x.String()
850         if mayCombine(p.lastTok, s[0]) {
851             // the previous and the curr
852             // separated by a blank othe
853             // into a different incorrec
854             // (except for token.INT fol
855             // should never happen becau
856             // of via binary expression
857             if len(p.wsbuff) != 0 {
858                 p.internalError("whi
859             }
860             p.wsbuff = p.wsbuff[0:1]
861             p.wsbuff[0] = ' '
862         }
863         data = s
864         // some keywords followed by a newli
865         switch x {
866         case token.BREAK, token.CONTINUE, to
867             token.INC, token.DEC, token.
868             implicitSemi = true
869         }
870         p.lastTok = x
871
872     case token.Pos:
873         if x.IsValid() {
874             p.pos = p.posFor(x) // accur
875         }
876         continue
877
878     case string:
879         // incorrect AST - print error messa
880         data = x

```

```

881         isLit = true
882         impliedSemi = true
883         p.lastTok = token.STRING
884
885     default:
886         fmt.Fprintf(os.Stderr, "print: unsup
887         panic("go/printer type")
888     }
889     // data != ""
890
891     next := p.pos // estimated/accurate position
892     wroteNewline, droppedFF := p.flush(next, p.l
893
894     // intersperse extra newlines if present in
895     // if they don't cause extra semicolons (don
896     // flush as it will cause extra newlines at
897     if !p.impliedSemi {
898         n := nlimit(next.Line - p.pos.Line)
899         // don't exceed maxNewlines if we al
900         if wroteNewline && n == maxNewlines
901             n = maxNewlines - 1
902     }
903     if n > 0 {
904         ch := byte('\n')
905         if droppedFF {
906             ch = '\f' // use for
907         }
908         p.WriteByte(ch, n)
909         impliedSemi = false
910     }
911 }
912
913     p.writeString(next, data, isLit)
914     p.impliedSemi = impliedSemi
915 }
916 }
917
918 // commentBefore returns true iff the current comment group
919 // before the next position in the source code and printing
920 // not introduce implicit semicolons.
921 //
922 func (p *printer) commentBefore(next token.Position) (result
923     return p.commentOffset < next.Offset && (!p.impliedS
924 }
925
926 // flush prints any pending comments and whitespace occurrin
927 // before the position of the next token tok. The flush resu
928 // if a newline was written or if a formfeed was dropped fro
929 // buffer.

```

```

930 //
931 func (p *printer) flush(next token.Position, tok token.Token
932     if p.commentBefore(next) {
933         // if there are comments before the next ite
934         wroteNewline, droppedFF = p.intersperseComme
935     } else {
936         // otherwise, write any leftover whitespace
937         p.writeWhitespace(len(p.wsbuf))
938     }
939     return
940 }
941
942 // getNode returns the ast.CommentGroup associated with n, i
943 func getDoc(n ast.Node) *ast.CommentGroup {
944     switch n := n.(type) {
945     case *ast.Field:
946         return n.Doc
947     case *ast.ImportSpec:
948         return n.Doc
949     case *ast.ValueSpec:
950         return n.Doc
951     case *ast.TypeSpec:
952         return n.Doc
953     case *ast.GenDecl:
954         return n.Doc
955     case *ast.FuncDecl:
956         return n.Doc
957     case *ast.File:
958         return n.Doc
959     }
960     return nil
961 }
962
963 func (p *printer) printNode(node interface{}) error {
964     // unpack *CommentedNode, if any
965     var comments []*ast.CommentGroup
966     if cnode, ok := node.(*CommentedNode); ok {
967         node = cnode.Node
968         comments = cnode.Comments
969     }
970
971     if comments != nil {
972         // commented node - restrict comment list to
973         n, ok := node.(ast.Node)
974         if !ok {
975             goto unsupported
976         }
977         beg := n.Pos()
978         end := n.End()
979         // if the node has associated documentation,

```

```

980         // include that commentgroup in the range
981         // (the comment list is sorted in the order
982         // of the comment appearance in the source c
983         if doc := getDoc(n); doc != nil {
984             beg = doc.Pos()
985         }
986         // token.Pos values are global offsets, we c
987         // compare them directly
988         i := 0
989         for i < len(comments) && comments[i].End() <
990             i++
991         }
992         j := i
993         for j < len(comments) && comments[j].Pos() <
994             j++
995         }
996         if i < j {
997             p.comments = comments[i:j]
998         }
999     } else if n, ok := node.(*ast.File); ok {
1000         // use ast.File comments, if any
1001         p.comments = n.Comments
1002     }
1003
1004     // if there are no comments, use node comments
1005     p.useNodeComments = p.comments == nil
1006
1007     // get comments ready for use
1008     p.nextComment()
1009
1010     // format node
1011     switch n := node.(type) {
1012     case ast.Expr:
1013         p.expr(n)
1014     case ast.Stmt:
1015         // A labeled statement will un-indent to pos
1016         // label. Set indent to 1 so we don't get in
1017         if _, labeledStmt := n.(*ast.LabeledStmt); l
1018             p.indent = 1
1019         }
1020         p.stmt(n, false)
1021     case ast.Decl:
1022         p.decl(n)
1023     case ast.Spec:
1024         p.spec(n, 1, false)
1025     case *ast.File:
1026         p.file(n)
1027     default:
1028         goto unsupported

```

```

1029         }
1030
1031         return nil
1032
1033     unsupported:
1034         return fmt.Errorf("go/printer: unsupported node type
1035     }
1036
1037     // -----
1038     // Trimmer
1039
1040     // A trimmer is an io.Writer filter for stripping tabwriter.
1041     // characters, trailing blanks and tabs, and for converting
1042     // and vtab characters into newlines and htabs (in case no t
1043     // is used). Text bracketed by tabwriter.Escape characters i
1044     // through unchanged.
1045     //
1046     type trimmer struct {
1047         output io.Writer
1048         state  int
1049         space  []byte
1050     }
1051
1052     // trimmer is implemented as a state machine.
1053     // It can be in one of the following states:
1054     const (
1055         inSpace = iota // inside space
1056         inEscape      // inside text bracketed by tabwrite
1057         inText        // inside text
1058     )
1059
1060     func (p *trimmer) resetSpace() {
1061         p.state = inSpace
1062         p.space = p.space[0:0]
1063     }
1064
1065     // Design note: It is tempting to eliminate extra blanks occ
1066     //             whitespace in this function as it could simp
1067     //             of the blanks logic in the node printing fun
1068     //             However, this would mess up any formatting d
1069     //             the tabwriter.
1070
1071     var aNewline = []byte("\n")
1072
1073     func (p *trimmer) Write(data []byte) (n int, err error) {
1074         // invariants:
1075         // p.state == inSpace:
1076         //     p.space is unwritten
1077         // p.state == inEscape, inText:

```

```

1078 //      data[m:n] is unwritten
1079 m := 0
1080 var b byte
1081 for n, b = range data {
1082     if b == '\v' {
1083         b = '\t' // convert to htab
1084     }
1085     switch p.state {
1086     case inSpace:
1087         switch b {
1088         case '\t', ' ':
1089             p.space = append(p.space, b)
1090         case '\n', '\f':
1091             p.resetSpace() // discard tr
1092             _, err = p.output.Write(aNew
1093     case tabwriter.Escape:
1094         _, err = p.output.Write(p.sp
1095         p.state = inEscape
1096         m = n + 1 // +1: skip tabwri
1097     default:
1098         _, err = p.output.Write(p.sp
1099         p.state = inText
1100         m = n
1101     }
1102     case inEscape:
1103         if b == tabwriter.Escape {
1104             _, err = p.output.Write(data
1105             p.resetSpace()
1106         }
1107     case inText:
1108         switch b {
1109         case '\t', ' ':
1110             _, err = p.output.Write(data
1111             p.resetSpace()
1112             p.space = append(p.space, b)
1113         case '\n', '\f':
1114             _, err = p.output.Write(data
1115             p.resetSpace()
1116             _, err = p.output.Write(aNew
1117     case tabwriter.Escape:
1118         _, err = p.output.Write(data
1119         p.state = inEscape
1120         m = n + 1 // +1: skip tabwri
1121     }
1122     default:
1123         panic("unreachable")
1124     }
1125     if err != nil {
1126         return
1127     }

```

```

1128     }
1129     n = len(data)
1130
1131     switch p.state {
1132     case inEscape, inText:
1133         _, err = p.output.Write(data[m:n])
1134         p.resetSpace()
1135     }
1136
1137     return
1138 }
1139
1140 // -----
1141 // Public interface
1142
1143 // A Mode value is a set of flags (or 0). They control print
1144 type Mode uint
1145
1146 const (
1147     RawFormat Mode = 1 << iota // do not use a tabwriter
1148     TabIndent           // use tabs for indentati
1149     UseSpaces           // use spaces instead of
1150     SourcePos           // emit //line comments t
1151 )
1152
1153 // A Config node controls the output of Fprint.
1154 type Config struct {
1155     Mode      Mode // default: 0
1156     Tabwidth  int  // default: 8
1157 }
1158
1159 // fprint implements Fprint and takes a nodesSizes map for s
1160 func (cfg *Config) fprint(output io.Writer, fset *token.File
1161     // print node
1162     var p printer
1163     p.init(cfg, fset, nodeSizes)
1164     if err = p.printNode(node); err != nil {
1165         return
1166     }
1167     // print outstanding comments
1168     p.implicitSemi = false // EOF acts like a newline
1169     p.flush(token.Position{Offset: infinity, Line: infin
1170
1171     // redirect output through a trimmer to eliminate tr
1172     // (Input to a tabwriter must be untrimmed since tra
1173     // formatting information. The tabwriter could provi
1174     // functionality but no tabwriter is used when RawFo
1175     output = &trimmer{output: output}
1176

```

```

1177         // redirect output through a tabwriter if necessary
1178         if cfg.Mode&RawFormat == 0 {
1179             minwidth := cfg.Tabwidth
1180
1181             padchar := byte('\t')
1182             if cfg.Mode&UseSpaces != 0 {
1183                 padchar = ' '
1184             }
1185
1186             twmode := tabwriter.DiscardEmptyColumns
1187             if cfg.Mode&TabIndent != 0 {
1188                 minwidth = 0
1189                 twmode |= tabwriter.TabIndent
1190             }
1191
1192             output = tabwriter.NewWriter(output, minwidth
1193         }
1194
1195         // write printer result via tabwriter/trimmer to out
1196         if _, err = output.Write(p.output); err != nil {
1197             return
1198         }
1199
1200         // flush tabwriter, if any
1201         if tw, _ := (output).(*tabwriter.Writer); tw != nil
1202             err = tw.Flush()
1203         }
1204
1205         return
1206     }
1207
1208     // A CommentedNode bundles an AST node and corresponding comments
1209     // It may be provided as argument to any of the Fprint functions
1210     //
1211     type CommentedNode struct {
1212         Node      interface{} // *ast.File, or ast.Expr, ast.
1213         Comments []*ast.CommentGroup
1214     }
1215
1216     // Fprint "pretty-prints" an AST node to output for a given
1217     // Position information is interpreted relative to the file
1218     // The node type must be *ast.File, *CommentedNode, or assigned
1219     // to ast.Expr, ast.Decl, ast.Spec, or ast.Stmt.
1220     //
1221     func (cfg *Config) Fprint(output io.Writer, fset *token.File
1222         return cfg.fprint(output, fset, node, make(map[ast.N
1223     }
1224
1225     // Fprint "pretty-prints" an AST node to output.

```

```
1226 // It calls Config.Fprint with default settings.
1227 //
1228 func Fprint(output io.Writer, fset *token.FileSet, node int)
1229     return (&Config{Tabwidth: 8}).Fprint(output, fset, n
1230 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/go/scanner/errors.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package scanner
6
7 import (
8     "fmt"
9     "go/token"
10    "io"
11    "sort"
12 )
13
14 // In an ErrorList, an error is represented by an *Error.
15 // The position Pos, if valid, points to the beginning of
16 // the offending token, and the error condition is described
17 // by Msg.
18 //
19 type Error struct {
20     Pos token.Position
21     Msg string
22 }
23
24 // Error implements the error interface.
25 func (e Error) Error() string {
26     if e.Pos.Filename != "" || e.Pos.IsValid() {
27         // don't print "<unknown position>"
28         // TODO(gri) reconsider the semantics of Pos
29         return e.Pos.String() + ": " + e.Msg
30     }
31     return e.Msg
32 }
33
34 // ErrorList is a list of *Errors.
35 // The zero value for an ErrorList is an empty ErrorList rea
36 //
37 type ErrorList []*Error
38
39 // Add adds an Error with given position and error message t
40 func (p *ErrorList) Add(pos token.Position, msg string) {
41     *p = append(*p, &Error{pos, msg})
42 }
```

```

42 }
43
44 // Reset resets an ErrorList to no errors.
45 func (p *ErrorList) Reset() { *p = (*p)[0:0] }
46
47 // ErrorList implements the sort Interface.
48 func (p ErrorList) Len() int      { return len(p) }
49 func (p ErrorList) Swap(i, j int) { p[i], p[j] = p[j], p[i] }
50
51 func (p ErrorList) Less(i, j int) bool {
52     e := &p[i].Pos
53     f := &p[j].Pos
54     // Note that it is not sufficient to simply compare
55     // the offsets do not reflect modified line informat
56     // comments).
57     if e.Filename < f.Filename {
58         return true
59     }
60     if e.Filename == f.Filename {
61         if e.Line < f.Line {
62             return true
63         }
64         if e.Line == f.Line {
65             return e.Column < f.Column
66         }
67     }
68     return false
69 }
70
71 // Sort sorts an ErrorList. *Error entries are sorted by pos
72 // other errors are sorted by error message, and before any
73 // entry.
74 //
75 func (p ErrorList) Sort() {
76     sort.Sort(p)
77 }
78
79 // RemoveMultiples sorts an ErrorList and removes all but th
80 func (p *ErrorList) RemoveMultiples() {
81     sort.Sort(p)
82     var last token.Position // initial last.Line is != a
83     i := 0
84     for _, e := range *p {
85         if e.Pos.Filename != last.Filename || e.Pos.
86             last = e.Pos
87             (*p)[i] = e
88             i++
89     }
90 }
91 (*p) = (*p)[0:i]

```

```

92 }
93
94 // An ErrorList implements the error interface.
95 func (p ErrorList) Error() string {
96     switch len(p) {
97     case 0:
98         return "no errors"
99     case 1:
100        return p[0].Error()
101    }
102    return fmt.Sprintf("%s (and %d more errors)", p[0],
103 }
104
105 // Err returns an error equivalent to this error list.
106 // If the list is empty, Err returns nil.
107 func (p ErrorList) Err() error {
108     if len(p) == 0 {
109         return nil
110     }
111     return p
112 }
113
114 // PrintError is a utility function that prints a list of er
115 // one error per line, if the err parameter is an ErrorList.
116 // it prints the err string.
117 //
118 func PrintError(w io.Writer, err error) {
119     if list, ok := err.(ErrorList); ok {
120         for _, e := range list {
121             fmt.Fprintf(w, "%s\n", e)
122         }
123     } else {
124         fmt.Fprintf(w, "%s\n", err)
125     }
126 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/go/scanner/scanner.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package scanner implements a scanner for Go source text.
6 // It takes a []byte as source which can then be tokenized
7 // through repeated calls to the Scan method.
8 //
9 package scanner
10
11 import (
12     "bytes"
13     "fmt"
14     "go/token"
15     "path/filepath"
16     "strconv"
17     "unicode"
18     "unicode/utf8"
19 )
20
21 // An ErrorHandler may be provided to Scanner.Init. If a syn
22 // encountered and a handler was installed, the handler is c
23 // position and an error message. The position points to the
24 // the offending token.
25 //
26 type ErrorHandler func(pos token.Position, msg string)
27
28 // A Scanner holds the scanner's internal state while proces
29 // a given text. It can be allocated as part of another dat
30 // structure but must be initialized via Init before use.
31 //
32 type Scanner struct {
33     // immutable state
34     file *token.File // source file handle
35     dir  string         // directory portion of file.Name()
36     src  []byte        // source
37     err  ErrorHandler   // error reporting; or nil
38     mode Mode          // scanning mode
39
40     // scanning state
41     ch rune // current character
```

```

42         offset      int // character offset
43         rdOffset   int // reading offset (position after cu
44         lineOffset int // current line offset
45         insertSemi bool // insert a semicolon before next ne
46
47         // public state - ok to modify
48         ErrorCount int // number of errors encountered
49     }
50
51     // Read the next Unicode char into s.ch.
52     // s.ch < 0 means end-of-file.
53     //
54     func (s *Scanner) next() {
55         if s.rdOffset < len(s.src) {
56             s.offset = s.rdOffset
57             if s.ch == '\n' {
58                 s.lineOffset = s.offset
59                 s.file.AddLine(s.offset)
60             }
61             r, w := rune(s.src[s.rdOffset]), 1
62             switch {
63             case r == 0:
64                 s.error(s.offset, "illegal character
65             case r >= 0x80:
66                 // not ASCII
67                 r, w = utf8.DecodeRune(s.src[s.rdOff
68                 if r == utf8.RuneError && w == 1 {
69                     s.error(s.offset, "illegal U
70                 }
71             }
72             s.rdOffset += w
73             s.ch = r
74         } else {
75             s.offset = len(s.src)
76             if s.ch == '\n' {
77                 s.lineOffset = s.offset
78                 s.file.AddLine(s.offset)
79             }
80             s.ch = -1 // eof
81         }
82     }
83
84     // A mode value is set of flags (or 0).
85     // They control scanner behavior.
86     //
87     type Mode uint
88
89     const (
90         ScanComments      Mode = 1 << iota // return comments
91         dontInsertSemis   // do not automatic

```

```

92 )
93
94 // Init prepares the scanner s to tokenize the text src by s
95 // scanner at the beginning of src. The scanner uses the fil
96 // for position information and it adds line information for
97 // It is ok to re-use the same file when re-scanning the sam
98 // line information which is already present is ignored. Ini
99 // panic if the file size does not match the src size.
100 //
101 // Calls to Scan will invoke the error handler err if they e
102 // syntax error and err is not nil. Also, for each error enc
103 // the Scanner field ErrorCount is incremented by one. The m
104 // determines how comments are handled.
105 //
106 // Note that Init may call err if there is an error in the f
107 // of the file.
108 //
109 func (s *Scanner) Init(file *token.File, src []byte, err Err
110 // Explicitly initialize all fields since a scanner
111 if file.Size() != len(src) {
112     panic(fmt.Sprintf("file size (%d) does not m
113 }
114     s.file = file
115     s.dir, _ = filepath.Split(file.Name())
116     s.src = src
117     s.err = err
118     s.mode = mode
119
120     s.ch = ' '
121     s.offset = 0
122     s.rdOffset = 0
123     s.lineOffset = 0
124     s.insertSemi = false
125     s.ErrorCount = 0
126
127     s.next()
128 }
129
130 func (s *Scanner) error(off int, msg string) {
131     if s.err != nil {
132         s.err(s.file.Position(s.file.Pos(off)), msg
133     }
134     s.ErrorCount++
135 }
136
137 var prefix = []byte("//line ")
138
139 func (s *Scanner) interpretLineComment(text []byte) {
140     if bytes.HasPrefix(text, prefix) {

```

```

141         // get filename and line number, if any
142         if i := bytes.LastIndex(text, []byte{' ':'\n'});
143             if line, err := strconv.Atoi(string(
144                 // valid //line filename:lin
145                 filename := filepath.Clean(s
146                 if !filepath.IsAbs(filename)
147                     // make filename rel
148                     filename = filepath.
149             )
150             // update scanner position
151             s.file.AddLineInfo(s.lineOff
152         }
153     }
154 }
155 }
156
157 func (s *Scanner) scanComment() string {
158     // initial '/' already consumed; s.ch == '/' || s.ch
159     offs := s.offset - 1 // position of initial '/'
160
161     if s.ch == '/' {
162         //-style comment
163         s.next()
164         for s.ch != '\n' && s.ch >= 0 {
165             s.next()
166         }
167         if offs == s.lineOffset {
168             // comment starts at the beginning o
169             s.interpretLineComment(s.src[offs:s.
170         }
171         goto exit
172     }
173
174     /*-style comment */
175     s.next()
176     for s.ch >= 0 {
177         ch := s.ch
178         s.next()
179         if ch == '*' && s.ch == '/' {
180             s.next()
181             goto exit
182         }
183     }
184
185     s.error(offs, "comment not terminated")
186
187 exit:
188     return string(s.src[offs:s.offset])
189 }

```

```

190
191 func (s *Scanner) findLineEnd() bool {
192     // initial '/' already consumed
193
194     defer func(off int) {
195         // reset scanner state to where it was upon
196         s.ch = '/'
197         s.offset = off
198         s.rdOffset = off + 1
199         s.next() // consume initial '/' again
200     }(s.offset - 1)
201
202     // read ahead until a newline, EOF, or non-comment token
203     for s.ch == '/' || s.ch == '*' {
204         if s.ch == '/' {
205             // -style comment always contains a newline
206             return true
207         }
208         /*-style comment: look for newline */
209         s.next()
210         for s.ch >= 0 {
211             ch := s.ch
212             if ch == '\n' {
213                 return true
214             }
215             s.next()
216             if ch == '*' && s.ch == '/' {
217                 s.next()
218                 break
219             }
220         }
221         s.skipWhitespace() // s.insertSemi is set
222         if s.ch < 0 || s.ch == '\n' {
223             return true
224         }
225         if s.ch != '/' {
226             // non-comment token
227             return false
228         }
229         s.next() // consume '/'
230     }
231
232     return false
233 }
234
235 func isLetter(ch rune) bool {
236     return 'a' <= ch && ch <= 'z' || 'A' <= ch && ch <=
237 }
238
239 func isDigit(ch rune) bool {

```

```

240         return '0' <= ch && ch <= '9' || ch >= 0x80 && unico
241     }
242
243     func (s *Scanner) scanIdentifier() string {
244         offs := s.offset
245         for isLetter(s.ch) || isDigit(s.ch) {
246             s.next()
247         }
248         return string(s.src[offs:s.offset])
249     }
250
251     func digitVal(ch rune) int {
252         switch {
253         case '0' <= ch && ch <= '9':
254             return int(ch - '0')
255         case 'a' <= ch && ch <= 'f':
256             return int(ch - 'a' + 10)
257         case 'A' <= ch && ch <= 'F':
258             return int(ch - 'A' + 10)
259         }
260         return 16 // larger than any legal digit val
261     }
262
263     func (s *Scanner) scanMantissa(base int) {
264         for digitVal(s.ch) < base {
265             s.next()
266         }
267     }
268
269     func (s *Scanner) scanNumber(seenDecimalPoint bool) (token.T
270     // digitVal(s.ch) < 10
271     offs := s.offset
272     tok := token.INT
273
274     if seenDecimalPoint {
275         offs--
276         tok = token.FLOAT
277         s.scanMantissa(10)
278         goto exponent
279     }
280
281     if s.ch == '0' {
282         // int or float
283         offs := s.offset
284         s.next()
285         if s.ch == 'x' || s.ch == 'X' {
286             // hexadecimal int
287             s.next()
288             s.scanMantissa(16)

```

```

289             if s.offset-offs <= 2 {
290                 // only scanned "0x" or "0X"
291                 s.error(offsets, "illegal hexad
292             }
293         } else {
294             // octal int or float
295             seenDecimalDigit := false
296             s.scanMantissa(8)
297             if s.ch == '8' || s.ch == '9' {
298                 // illegal octal int or floa
299                 seenDecimalDigit = true
300                 s.scanMantissa(10)
301             }
302             if s.ch == '.' || s.ch == 'e' || s.c
303                 goto fraction
304             }
305             // octal int
306             if seenDecimalDigit {
307                 s.error(offsets, "illegal octal
308             }
309         }
310         goto exit
311     }
312
313     // decimal int or float
314     s.scanMantissa(10)
315
316 fraction:
317     if s.ch == '.' {
318         tok = token.FLOAT
319         s.next()
320         s.scanMantissa(10)
321     }
322
323 exponent:
324     if s.ch == 'e' || s.ch == 'E' {
325         tok = token.FLOAT
326         s.next()
327         if s.ch == '-' || s.ch == '+' {
328             s.next()
329         }
330         s.scanMantissa(10)
331     }
332
333     if s.ch == 'i' {
334         tok = token.IMAG
335         s.next()
336     }
337

```

```

338 exit:
339     return tok, string(s.src[offs:s.offset])
340 }
341
342 func (s *Scanner) scanEscape(quote rune) {
343     offs := s.offset
344
345     var i, base, max uint32
346     switch s.ch {
347     case 'a', 'b', 'f', 'n', 'r', 't', 'v', '\\\\', quote:
348         s.next()
349         return
350     case '0', '1', '2', '3', '4', '5', '6', '7':
351         i, base, max = 3, 8, 255
352     case 'x':
353         s.next()
354         i, base, max = 2, 16, 255
355     case 'u':
356         s.next()
357         i, base, max = 4, 16, unicode.MaxRune
358     case 'U':
359         s.next()
360         i, base, max = 8, 16, unicode.MaxRune
361     default:
362         s.next() // always make progress
363         s.error(offs, "unknown escape sequence")
364         return
365     }
366
367     var x uint32
368     for ; i > 0 && s.ch != quote && s.ch >= 0; i-- {
369         d := uint32(digitVal(s.ch))
370         if d >= base {
371             s.error(s.offset, "illegal character")
372             break
373         }
374         x = x*base + d
375         s.next()
376     }
377     // in case of an error, consume remaining chars
378     for ; i > 0 && s.ch != quote && s.ch >= 0; i-- {
379         s.next()
380     }
381     if x > max || 0xd800 <= x && x < 0xe000 {
382         s.error(offs, "escape sequence is invalid Un")
383     }
384 }
385
386 func (s *Scanner) scanChar() string {
387     // '\\' opening already consumed

```

```

388     offs := s.offset - 1
389
390     n := 0
391     for s.ch != '\\' {
392         ch := s.ch
393         n++
394         s.next()
395         if ch == '\n' || ch < 0 {
396             s.error(offs, "character literal not
397                 n = 1
398                 break
399         }
400         if ch == '\\\ ' {
401             s.scanEscape('\ ' )
402         }
403     }
404
405     s.next()
406
407     if n != 1 {
408         s.error(offs, "illegal character literal")
409     }
410
411     return string(s.src[offs:s.offset])
412 }
413
414 func (s *Scanner) scanString() string {
415     // '"' opening already consumed
416     offs := s.offset - 1
417
418     for s.ch != '"' {
419         ch := s.ch
420         s.next()
421         if ch == '\n' || ch < 0 {
422             s.error(offs, "string not terminated
423                 break
424         }
425         if ch == '\\\ ' {
426             s.scanEscape('\ ' )
427         }
428     }
429
430     s.next()
431
432     return string(s.src[offs:s.offset])
433 }
434
435 func stripCR(b []byte) []byte {
436     c := make([]byte, len(b))

```

```

437         i := 0
438         for _, ch := range b {
439             if ch != '\r' {
440                 c[i] = ch
441                 i++
442             }
443         }
444         return c[:i]
445     }
446
447     func (s *Scanner) scanRawString() string {
448         // `` opening already consumed
449         offs := s.offset - 1
450
451         hasCR := false
452         for s.ch != `` {
453             ch := s.ch
454             s.next()
455             if ch == '\r' {
456                 hasCR = true
457             }
458             if ch < 0 {
459                 s.error(offs, "string not terminated
460                 break
461             }
462         }
463
464         s.next()
465
466         lit := s.src[offs:s.offset]
467         if hasCR {
468             lit = stripCR(lit)
469         }
470
471         return string(lit)
472     }
473
474     func (s *Scanner) skipWhitespace() {
475         for s.ch == ' ' || s.ch == '\t' || s.ch == '\n' && !
476             s.next()
477     }
478 }
479
480 // Helper functions for scanning multi-byte tokens such as >
481 // Different routines recognize different length tok_i based
482 // of ch_i. If a token ends in '=', the result is tok1 or to
483 // respectively. Otherwise, the result is tok0 if there was
484 // matching character, or tok2 if the matching character was
485

```

```

486 func (s *Scanner) switch2(tok0, tok1 token.Token) token.Toke
487     if s.ch == '=' {
488         s.next()
489         return tok1
490     }
491     return tok0
492 }
493
494 func (s *Scanner) switch3(tok0, tok1 token.Token, ch2 rune,
495     if s.ch == '=' {
496         s.next()
497         return tok1
498     }
499     if s.ch == ch2 {
500         s.next()
501         return tok2
502     }
503     return tok0
504 }
505
506 func (s *Scanner) switch4(tok0, tok1 token.Token, ch2 rune,
507     if s.ch == '=' {
508         s.next()
509         return tok1
510     }
511     if s.ch == ch2 {
512         s.next()
513         if s.ch == '=' {
514             s.next()
515             return tok3
516         }
517         return tok2
518     }
519     return tok0
520 }
521
522 // Scan scans the next token and returns the token position,
523 // and its literal string if applicable. The source end is i
524 // token.EOF.
525 //
526 // If the returned token is a literal (token.IDENT, token.IN
527 // token.IMAG, token.CHAR, token.STRING) or token.COMMENT, t
528 // has the corresponding value.
529 //
530 // If the returned token is token.SEMICOLON, the correspondi
531 // literal string is ";" if the semicolon was present in the
532 // and "\n" if the semicolon was inserted because of a newli
533 // at EOF.
534 //
535 // If the returned token is token.ILLEGAL, the literal strin

```

```

536 // offending character.
537 //
538 // In all other cases, Scan returns an empty literal string.
539 //
540 // For more tolerant parsing, Scan will return a valid token
541 // possible even if a syntax error was encountered. Thus, ev
542 // if the resulting token sequence contains no illegal token
543 // a client may not assume that no error occurred. Instead i
544 // must check the scanner's ErrorCount or the number of call
545 // of the error handler, if there was one installed.
546 //
547 // Scan adds line information to the file added to the file
548 // set with Init. Token positions are relative to that file
549 // and thus relative to the file set.
550 //
551 func (s *Scanner) Scan() (pos token.Pos, tok token.Token, li
552 scanAgain:
553     s.skipWhitespace()
554
555     // current token start
556     pos = s.file.Pos(s.offset)
557
558     // determine token value
559     insertSemi := false
560     switch ch := s.ch; {
561     case isLetter(ch):
562         lit = s.scanIdentifier()
563         tok = token.Lookup(lit)
564         switch tok {
565         case token.IDENT, token.BREAK, token.CONTINU
566             insertSemi = true
567         }
568     case digitVal(ch) < 10:
569         insertSemi = true
570         tok, lit = s.scanNumber(false)
571     default:
572         s.next() // always make progress
573         switch ch {
574         case -1:
575             if s.insertSemi {
576                 s.insertSemi = false // EOF
577                 return pos, token.SEMICOLON,
578             }
579             tok = token.EOF
580         case '\n':
581             // we only reach here if s.insertSem
582             // set in the first place and exited
583             // from s.skipWhitespace()
584             s.insertSemi = false // newline cons

```

```

585         return pos, token.SEMICOLON, "\n"
586     case '"':
587         insertSemi = true
588         tok = token.STRING
589         lit = s.scanString()
590     case '\':
591         insertSemi = true
592         tok = token.CHAR
593         lit = s.scanChar()
594     case '`':
595         insertSemi = true
596         tok = token.STRING
597         lit = s.scanRawString()
598     case ':':
599         tok = s.switch2(token.COLON, token.D
600     case '.':
601         if digitVal(s.ch) < 10 {
602             insertSemi = true
603             tok, lit = s.scanNumber(true
604         } else if s.ch == '.' {
605             s.next()
606             if s.ch == '.' {
607                 s.next()
608                 tok = token.ELLIPSIS
609             }
610         } else {
611             tok = token.PERIOD
612         }
613     case ',':
614         tok = token.COMMA
615     case ';':
616         tok = token.SEMICOLON
617         lit = ";"
618     case '(':
619         tok = token.LPAREN
620     case ')':
621         insertSemi = true
622         tok = token.RPAREN
623     case '[':
624         tok = token.LBRACK
625     case ']':
626         insertSemi = true
627         tok = token.RBRACK
628     case '{':
629         tok = token.LBRACE
630     case '}':
631         insertSemi = true
632         tok = token.RBRACE
633     case '+':

```

```

634         tok = s.switch3(token.ADD, token.ADD
635         if tok == token.INC {
636             insertSemi = true
637         }
638     case '-':
639         tok = s.switch3(token.SUB, token.SUB
640         if tok == token.DEC {
641             insertSemi = true
642         }
643     case '*':
644         tok = s.switch2(token.MUL, token.MUL
645     case '/':
646         if s.ch == '/' || s.ch == '*' {
647             // comment
648             if s.insertSemi && s.findLin
649                 // reset position to
650                 s.ch = '/'
651                 s.offset = s.file.Of
652                 s.rdOffset = s.offse
653                 s.insertSemi = false
654                 return pos, token.SE
655         }
656         lit = s.scanComment()
657         if s.mode&ScanComments == 0
658             // skip comment
659             s.insertSemi = false
660             goto scanAgain
661         }
662         tok = token.COMMENT
663     } else {
664         tok = s.switch2(token.QUO, t
665     }
666     case '%':
667         tok = s.switch2(token.REM, token.REM
668     case '^':
669         tok = s.switch2(token.XOR, token.XOR
670     case '<':
671         if s.ch == '-' {
672             s.next()
673             tok = token.ARROW
674         } else {
675             tok = s.switch4(token.LSS, t
676         }
677     case '>':
678         tok = s.switch4(token.GTR, token.GEQ
679     case '=':
680         tok = s.switch2(token.ASSIGN, token.
681     case '!':
682         tok = s.switch2(token.NOT, token.NEQ
683     case '&':

```

```

684             if s.ch == '^' {
685                 s.next()
686                 tok = s.switch2(token.AND_NC
687             } else {
688                 tok = s.switch3(token.AND, t
689             }
690         case '|':
691             tok = s.switch3(token.OR, token.OR_A
692         default:
693             s.error(s.file.Offset(pos), fmt.Spri
694             insertSemi = s.insertSemi // preserv
695             tok = token.ILLEGAL
696             lit = string(ch)
697         }
698     }
699     if s.mode&dontInsertSemis == 0 {
700         s.insertSemi = insertSemi
701     }
702
703     return
704 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/go/token/position.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // TODO(gri) consider making this a separate package outside
6
7 package token
8
9 import (
10     "fmt"
11     "sort"
12     "sync"
13 )
14
15 // -----
16 // Positions
17
18 // Position describes an arbitrary source position
19 // including the file, line, and column location.
20 // A Position is valid if the line number is > 0.
21 //
22 type Position struct {
23     Filename string // filename, if any
24     Offset   int    // offset, starting at 0
25     Line     int    // line number, starting at 1
26     Column   int    // column number, starting at 1 (cha
27 }
28
29 // IsValid returns true if the position is valid.
30 func (pos *Position) IsValid() bool { return pos.Line > 0 }
31
32 // String returns a string in one of several forms:
33 //
34 //     file:line:column    valid position with file name
35 //     line:column         valid position without file name
36 //     file                 invalid position with file name
37 //     -                   invalid position without file na
38 //
39 func (pos Position) String() string {
40     s := pos.Filename
41     if pos.IsValid() {
```

```

42         if s != "" {
43             s += ":"
44         }
45         s += fmt.Sprintf("%d:%d", pos.Line, pos.Colu
46     }
47     if s == "" {
48         s = "-"
49     }
50     return s
51 }
52
53 // Pos is a compact encoding of a source position within a f
54 // It can be converted into a Position for a more convenient
55 // larger, representation.
56 //
57 // The Pos value for a given file is a number in the range [
58 // where base and size are specified when adding the file to
59 // AddFile.
60 //
61 // To create the Pos value for a specific source offset, fir
62 // the respective file to the current file set (via FileSet.
63 // and then call File.Pos(offset) for that file. Given a Pos
64 // for a specific file set fset, the corresponding Position
65 // obtained by calling fset.Position(p).
66 //
67 // Pos values can be compared directly with the usual compar
68 // If two Pos values p and q are in the same file, comparing
69 // equivalent to comparing the respective source file offset
70 // are in different files, p < q is true if the file implied
71 // to the respective file set before the file implied by q.
72 //
73 type Pos int
74
75 // The zero value for Pos is NoPos; there is no file and lin
76 // associated with it, and NoPos().IsValid() is false. NoPos
77 // smaller than any other Pos value. The corresponding Posit
78 // for NoPos is the zero value for Position.
79 //
80 const NoPos Pos = 0
81
82 // IsValid returns true if the position is valid.
83 func (p Pos) IsValid() bool {
84     return p != NoPos
85 }
86
87 // -----
88 // File
89
90 // A File is a handle for a file belonging to a FileSet.
91 // A File has a name, size, and line offset table.

```

```

92 //
93 type File struct {
94     set *FileSet
95     name string // file name as provided to AddFile
96     base int    // Pos value range for this file is [bas
97     size int    // file size as provided to AddFile
98
99     // lines and infos are protected by set.mutex
100    lines []int
101    infos []lineInfo
102 }
103
104 // Name returns the file name of file f as registered with A
105 func (f *File) Name() string {
106     return f.name
107 }
108
109 // Base returns the base offset of file f as registered with
110 func (f *File) Base() int {
111     return f.base
112 }
113
114 // Size returns the size of file f as registered with AddFil
115 func (f *File) Size() int {
116     return f.size
117 }
118
119 // LineCount returns the number of lines in file f.
120 func (f *File) LineCount() int {
121     f.set.mutex.RLock()
122     n := len(f.lines)
123     f.set.mutex.RUnlock()
124     return n
125 }
126
127 // AddLine adds the line offset for a new line.
128 // The line offset must be larger than the offset for the pr
129 // and smaller than the file size; otherwise the line offset
130 //
131 func (f *File) AddLine(offset int) {
132     f.set.mutex.Lock()
133     if i := len(f.lines); (i == 0 || f.lines[i-1] < offs
134         f.lines = append(f.lines, offset)
135     }
136     f.set.mutex.Unlock()
137 }
138
139 // SetLines sets the line offsets for a file and returns tru
140 // The line offsets are the offsets of the first character o

```

```

141 // for instance for the content "ab\nc\n" the line offsets a
142 // An empty file has an empty line offset table.
143 // Each line offset must be larger than the offset for the p
144 // and smaller than the file size; otherwise SetLines fails
145 // false.
146 //
147 func (f *File) SetLines(lines []int) bool {
148     // verify validity of lines table
149     size := f.size
150     for i, offset := range lines {
151         if i > 0 && offset <= lines[i-1] || size <=
152             return false
153     }
154 }
155
156 // set lines table
157 f.set.mutex.Lock()
158 f.lines = lines
159 f.set.mutex.Unlock()
160 return true
161 }
162
163 // SetLinesForContent sets the line offsets for the given fi
164 func (f *File) SetLinesForContent(content []byte) {
165     var lines []int
166     line := 0
167     for offset, b := range content {
168         if line >= 0 {
169             lines = append(lines, line)
170         }
171         line = -1
172         if b == '\n' {
173             line = offset + 1
174         }
175     }
176
177     // set lines table
178     f.set.mutex.Lock()
179     f.lines = lines
180     f.set.mutex.Unlock()
181 }
182
183 // A lineInfo object describes alternative file and line num
184 // information (such as provided via a //line comment in a .
185 // file) for a given file offset.
186 type lineInfo struct {
187     // fields are exported to make them accessible to go
188     Offset int
189     Filename string

```

```

190         Line    int
191     }
192
193     // AddLineInfo adds alternative file and line number informa
194     // a given file offset. The offset must be larger than the o
195     // the previously added alternative line info and smaller th
196     // file size; otherwise the information is ignored.
197     //
198     // AddLineInfo is typically used to register alternative pos
199     // information for //line filename:line comments in source f
200     //
201     func (f *File) AddLineInfo(offset int, filename string, line
202         f.set.mutex.Lock()
203         if i := len(f.infos); i == 0 || f.infos[i-1].Offset
204             f.infos = append(f.infos, lineInfo{offset, f
205         }
206         f.set.mutex.Unlock()
207     }
208
209     // Pos returns the Pos value for the given file offset;
210     // the offset must be <= f.Size().
211     // f.Pos(f.Offset(p)) == p.
212     //
213     func (f *File) Pos(offset int) Pos {
214         if offset > f.size {
215             panic("illegal file offset")
216         }
217         return Pos(f.base + offset)
218     }
219
220     // Offset returns the offset for the given file position p;
221     // p must be a valid Pos value in that file.
222     // f.Offset(f.Pos(offset)) == offset.
223     //
224     func (f *File) Offset(p Pos) int {
225         if int(p) < f.base || int(p) > f.base+f.size {
226             panic("illegal Pos value")
227         }
228         return int(p) - f.base
229     }
230
231     // Line returns the line number for the given file position
232     // p must be a Pos value in that file or NoPos.
233     //
234     func (f *File) Line(p Pos) int {
235         // TODO(gri) this can be implemented much more effic
236         return f.Position(p).Line
237     }
238
239     func searchLineInfos(a []lineInfo, x int) int {

```

```

240         return sort.Search(len(a), func(i int) bool { return
241     }
242
243 // info returns the file name, line, and column number for a
244 func (f *File) info(offset int) (filename string, line, colu
245     filename = f.name
246     if i := searchInts(f.lines, offset); i >= 0 {
247         line, column = i+1, offset-f.lines[i]+1
248     }
249     if len(f.infos) > 0 {
250         // almost no files have extra line infos
251         if i := searchLineInfos(f.infos, offset); i
252             alt := &f.infos[i]
253             filename = alt.FileName
254             if i := searchInts(f.lines, alt.Offs
255                 line += alt.Line - i - 1
256             }
257         }
258     }
259     return
260 }
261
262 func (f *File) position(p Pos) (pos Position) {
263     offset := int(p) - f.base
264     pos.Offset = offset
265     pos.FileName, pos.Line, pos.Column = f.info(offset)
266     return
267 }
268
269 // Position returns the Position value for the given file po
270 // p must be a Pos value in that file or NoPos.
271 //
272 func (f *File) Position(p Pos) (pos Position) {
273     if p != NoPos {
274         if int(p) < f.base || int(p) > f.base+f.size
275             panic("illegal Pos value")
276         }
277         pos = f.position(p)
278     }
279     return
280 }
281
282 // -----
283 // FileSet
284
285 // A FileSet represents a set of source files.
286 // Methods of file sets are synchronized; multiple goroutine
287 // may invoke them concurrently.
288 //

```

```

289 type FileSet struct {
290     mutex sync.RWMutex // protects the file set
291     base int           // base offset for the next file
292     files []*File      // list of files in the order add
293     last *File         // cache of last file looked up
294 }
295
296 // NewFileSet creates a new file set.
297 func NewFileSet() *FileSet {
298     s := new(FileSet)
299     s.base = 1 // 0 == NoPos
300     return s
301 }
302
303 // Base returns the minimum base offset that must be provide
304 // AddFile when adding the next file.
305 //
306 func (s *FileSet) Base() int {
307     s.mutex.RLock()
308     b := s.base
309     s.mutex.RUnlock()
310     return b
311 }
312 }
313
314 // AddFile adds a new file with a given filename, base offse
315 // to the file set s and returns the file. Multiple files ma
316 // name. The base offset must not be smaller than the FileSe
317 // size must not be negative.
318 //
319 // Adding the file will set the file set's Base() value to b
320 // as the minimum base value for the next file. The followin
321 // exists between a Pos value p for a given file offset offs
322 //
323 //      int(p) = base + offs
324 //
325 // with offs in the range [0, size] and thus p in the range
326 // For convenience, File.Pos may be used to create file-spec
327 // values from a file offset.
328 //
329 func (s *FileSet) AddFile(filename string, base, size int) *
330     s.mutex.Lock()
331     defer s.mutex.Unlock()
332     if base < s.base || size < 0 {
333         panic("illegal base or size")
334     }
335     // base >= s.base && size >= 0
336     f := &File{s, filename, base, size, []int{0}, nil}
337     base += size + 1 // +1 because EOF also has a positi

```

```

338         if base < 0 {
339             panic("token.Pos offset overflow (> 2G of so
340         }
341         // add the file to the file set
342         s.base = base
343         s.files = append(s.files, f)
344         s.last = f
345         return f
346     }
347
348     // Iterate calls f for the files in the file set in the orde
349     // until f returns false.
350     //
351     func (s *FileSet) Iterate(f func(*File) bool) {
352         for i := 0; ; i++ {
353             var file *File
354             s.mutex.RLock()
355             if i < len(s.files) {
356                 file = s.files[i]
357             }
358             s.mutex.RUnlock()
359             if file == nil || !f(file) {
360                 break
361             }
362         }
363     }
364
365     func searchFiles(a []*File, x int) int {
366         return sort.Search(len(a), func(i int) bool { return
367     }
368
369     func (s *FileSet) file(p Pos) *File {
370         // common case: p is in last file
371         if f := s.last; f != nil && f.base <= int(p) && int(
372             return f
373     }
374     // p is not in last file - search all files
375     if i := searchFiles(s.files, int(p)); i >= 0 {
376         f := s.files[i]
377         // f.base <= int(p) by definition of searchF
378         if int(p) <= f.base+f.size {
379             s.last = f
380             return f
381         }
382     }
383     return nil
384 }
385
386 // File returns the file that contains the position p.
387 // If no such file is found (for instance for p == NoPos),

```

```

388 // the result is nil.
389 //
390 func (s *FileSet) File(p Pos) (f *File) {
391     if p != NoPos {
392         s.mutex.RLock()
393         f = s.file(p)
394         s.mutex.RUnlock()
395     }
396     return
397 }
398
399 // Position converts a Pos in the fileset into a general Pos
400 func (s *FileSet) Position(p Pos) (pos Position) {
401     if p != NoPos {
402         s.mutex.RLock()
403         if f := s.file(p); f != nil {
404             pos = f.position(p)
405         }
406         s.mutex.RUnlock()
407     }
408     return
409 }
410
411 // -----
412 // Helper functions
413
414 func searchInts(a []int, x int) int {
415     // This function body is a manually inlined version
416     //
417     // return sort.Search(len(a), func(i int) bool { r
418     //
419     // With better compiler optimizations, this may not
420     // future, but at the moment this change improves th
421     // benchmark performance by ~30%. This has a direct
422     // speed of gofmt and thus seems worthwhile (2011-04
423     // TODO(gri): Remove this when compilers have caught
424     i, j := 0, len(a)
425     for i < j {
426         h := i + (j-i)/2 // avoid overflow when comp
427         // i ≤ h < j
428         if a[h] <= x {
429             i = h + 1
430         } else {
431             j = h
432         }
433     }
434     return i - 1
435 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/go/token/serialize.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package token
6
7 type serializedFile struct {
8     // fields correspond 1:1 to fields with same (lower-
9     Name string
10    Base int
11    Size int
12    Lines []int
13    Infos []lineInfo
14 }
15
16 type serializedFileSet struct {
17    Base int
18    Files []serializedFile
19 }
20
21 // Read calls decode to deserialize a file set into s; s must
22 func (s *FileSet) Read(decode func(interface{}) error) error
23    var ss serializedFileSet
24    if err := decode(&ss); err != nil {
25        return err
26    }
27
28    s.mutex.Lock()
29    s.base = ss.Base
30    files := make([]*File, len(ss.Files))
31    for i := 0; i < len(ss.Files); i++ {
32        f := &ss.Files[i]
33        files[i] = &File{s, f.Name, f.Base, f.Size,
34    }
35    s.files = files
36    s.last = nil
37    s.mutex.Unlock()
38
39    return nil
40 }
41
```

```
42 // Write calls encode to serialize the file set s.
43 func (s *FileSet) Write(encode func(interface{})) error) erro
44     var ss serializedFileSet
45
46     s.mutex.Lock()
47     ss.Base = s.base
48     files := make([]serializedFile, len(s.files))
49     for i, f := range s.files {
50         files[i] = serializedFile{f.name, f.base, f.
51     }
52     ss.Files = files
53     s.mutex.Unlock()
54
55     return encode(ss)
56 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/go/token/token.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package token defines constants representing the lexical
6 // programming language and basic operations on tokens (prin
7 //
8 package token
9
10 import "strconv"
11
12 // Token is the set of lexical tokens of the Go programming
13 type Token int
14
15 // The list of tokens.
16 const (
17     // Special tokens
18     ILLEGAL Token = iota
19     EOF
20     COMMENT
21
22     literal_beg
23     // Identifiers and basic type literals
24     // (these tokens stand for classes of literals)
25     IDENT // main
26     INT   // 12345
27     FLOAT // 123.45
28     IMAG  // 123.45i
29     CHAR  // 'a'
30     STRING // "abc"
31     literal_end
32
33     operator_beg
34     // Operators and delimiters
35     ADD // +
36     SUB // -
37     MUL // *
38     QUO // /
39     REM // %
40
41     AND // &
42     OR  // |
43     XOR // ^
44     SHL // <<
```

```
45     SHR      // >>
46     AND_NOT // &^
47
48     ADD_ASSIGN // +=
49     SUB_ASSIGN // -=
50     MUL_ASSIGN // *=
51     QUO_ASSIGN // /=
52     REM_ASSIGN // %=
53
54     AND_ASSIGN // &=
55     OR_ASSIGN  // |=
56     XOR_ASSIGN // ^=
57     SHL_ASSIGN // <<=
58     SHR_ASSIGN // >>=
59     AND_NOT_ASSIGN // &^=
60
61     LAND // &&
62     LOR  // ||
63     ARROW // <-
64     INC  // ++
65     DEC  // --
66
67     EQL // ==
68     LSS // <
69     GTR // >
70     ASSIGN // =
71     NOT // !
72
73     NEQ // !=
74     LEQ // <=
75     GEQ // >=
76     DEFINE // :=
77     ELLIPSIS // ...
78
79     LPAREN // (
80     LBRACK // [
81     LBRACE // {
82     COMMA // ,
83     PERIOD // .
84
85     RPAREN // )
86     RBRACK // ]
87     RBRACE // }
88     SEMICOLON // ;
89     COLON // :
90     operator_end
91
92     keyword_beg
93     // Keywords
94     BREAK
```

```
95         CASE
96         CHAN
97         CONST
98         CONTINUE
99
100        DEFAULT
101        DEFER
102        ELSE
103        FALLTHROUGH
104        FOR
105
106        FUNC
107        GO
108        GOTO
109        IF
110        IMPORT
111
112        INTERFACE
113        MAP
114        PACKAGE
115        RANGE
116        RETURN
117
118        SELECT
119        STRUCT
120        SWITCH
121        TYPE
122        VAR
123        keyword_end
124    )
125
126    var tokens = [...]string{
127        ILLEGAL: "ILLEGAL",
128
129        EOF:      "EOF",
130        COMMENT: "COMMENT",
131
132        IDENT:  "IDENT",
133        INT:    "INT",
134        FLOAT:  "FLOAT",
135        IMAG:   "IMAG",
136        CHAR:   "CHAR",
137        STRING: "STRING",
138
139        ADD: "+",
140        SUB: "-",
141        MUL: "*",
142        QUO: "/",
143        REM: "%",
```

```

144
145     AND:      "&",
146     OR:       "|",
147     XOR:      "^",
148     SHL:      "<<",
149     SHR:      ">>",
150     AND_NOT:  "&^",
151
152     ADD_ASSIGN: "+=",
153     SUB_ASSIGN: "-=",
154     MUL_ASSIGN: "*=",
155     QUO_ASSIGN: "/=",
156     REM_ASSIGN: "%=",
157
158     AND_ASSIGN: "&=",
159     OR_ASSIGN:  "|=",
160     XOR_ASSIGN: "^=",
161     SHL_ASSIGN: "<<=",
162     SHR_ASSIGN: ">>=",
163     AND_NOT_ASSIGN: "&^=",
164
165     LAND:     "&&",
166     LOR:      "||",
167     ARROW:    "<- ",
168     INC:      "++",
169     DEC:      "--",
170
171     EQL:      "==",
172     LSS:      "<",
173     GTR:      ">",
174     ASSIGN:   "=",
175     NOT:      "!",
176
177     NEQ:      "!=",
178     LEQ:      "<=",
179     GEQ:      ">=",
180     DEFINE:   ":",
181     ELLIPSIS: "...",
182
183     LPAREN:  "(",
184     LBRACK:  "[",
185     LBRACE:  "{",
186     COMMA:   ",",
187     PERIOD:  ".",
188
189     RPAREN:  ")",
190     RBRACK:  "]",
191     RBRACE:  "}",
192     SEMICOLON: ";",

```

```

193         COLON:      ":",
194
195         BREAK:      "break",
196         CASE:       "case",
197         CHAN:       "chan",
198         CONST:      "const",
199         CONTINUE:   "continue",
200
201         DEFAULT:    "default",
202         DEFER:      "defer",
203         ELSE:       "else",
204         FALLTHROUGH: "fallthrough",
205         FOR:        "for",
206
207         FUNC:       "func",
208         GO:         "go",
209         GOTO:       "goto",
210         IF:         "if",
211         IMPORT:     "import",
212
213         INTERFACE:  "interface",
214         MAP:        "map",
215         PACKAGE:    "package",
216         RANGE:      "range",
217         RETURN:     "return",
218
219         SELECT:     "select",
220         STRUCT:     "struct",
221         SWITCH:     "switch",
222         TYPE:       "type",
223         VAR:        "var",
224     }
225
226     // String returns the string corresponding to the token tok.
227     // For operators, delimiters, and keywords the string is the
228     // token character sequence (e.g., for the token ADD, the st
229     // "+"). For all other tokens the string corresponds to the
230     // constant name (e.g. for the token IDENT, the string is "I
231     //
232     func (tok Token) String() string {
233         s := ""
234         if 0 <= tok && tok < Token(len(tokens)) {
235             s = tokens[tok]
236         }
237         if s == "" {
238             s = "token(" + strconv.Itoa(int(tok)) + ")"
239         }
240         return s
241     }
242

```

```

243 // A set of constants for precedence-based expression parsing
244 // Non-operators have lowest precedence, followed by operators
245 // starting with precedence 1 up to unary operators. The highest
246 // precedence corresponds to the "catch-all" precedence for
247 // selector, indexing, and other operator and delimiter tokens
248 //
249 const (
250     LowestPrec = 0 // non-operators
251     UnaryPrec  = 6
252     HighestPrec = 7
253 )
254
255 // Precedence returns the operator precedence of the binary
256 // operator op. If op is not a binary operator, the result
257 // is LowestPrecedence.
258 //
259 func (op Token) Precedence() int {
260     switch op {
261     case LOR:
262         return 1
263     case LAND:
264         return 2
265     case EQL, NEQ, LSS, LEQ, GTR, GEQ:
266         return 3
267     case ADD, SUB, OR, XOR:
268         return 4
269     case MUL, QUO, REM, SHL, SHR, AND, AND_NOT:
270         return 5
271     }
272     return LowestPrec
273 }
274
275 var keywords map[string]Token
276
277 func init() {
278     keywords = make(map[string]Token)
279     for i := keyword_beg + 1; i < keyword_end; i++ {
280         keywords[tokens[i]] = i
281     }
282 }
283
284 // Lookup maps an identifier to its keyword token or IDENT (
285 //
286 func Lookup(ident string) Token {
287     if tok, is_keyword := keywords[ident]; is_keyword {
288         return tok
289     }
290     return IDENT
291 }

```

```
292
293 // Predicates
294
295 // IsLiteral returns true for tokens corresponding to identi
296 // and basic type literals; it returns false otherwise.
297 //
298 func (tok Token) IsLiteral() bool { return literal_beg < tok
299
300 // IsOperator returns true for tokens corresponding to opera
301 // delimiters; it returns false otherwise.
302 //
303 func (tok Token) IsOperator() bool { return operator_beg < t
304
305 // IsKeyword returns true for tokens corresponding to keywor
306 // it returns false otherwise.
307 //
308 func (tok Token) IsKeyword() bool { return keyword_beg < tok
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/hash/hash.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package hash provides interfaces for hash functions.
6 package hash
7
8 import "io"
9
10 // Hash is the common interface implemented by all hash func
11 type Hash interface {
12     // Write adds more data to the running hash.
13     // It never returns an error.
14     io.Writer
15
16     // Sum appends the current hash to b and returns the
17     // It does not change the underlying hash state.
18     Sum(b []byte) []byte
19
20     // Reset resets the hash to one with zero bytes writ
21     Reset()
22
23     // Size returns the number of bytes Sum will return.
24     Size() int
25
26     // BlockSize returns the hash's underlying block siz
27     // The Write method must be able to accept any amoun
28     // of data, but it may operate more efficiently if a
29     // are a multiple of the block size.
30     BlockSize() int
31 }
32
33 // Hash32 is the common interface implemented by all 32-bit
34 type Hash32 interface {
35     Hash
36     Sum32() uint32
37 }
38
39 // Hash64 is the common interface implemented by all 64-bit
40 type Hash64 interface {
41     Hash
42     Sum64() uint64
43 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/hash/adler32/adler32.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package adler32 implements the Adler-32 checksum.
6 // Defined in RFC 1950:
7 //     Adler-32 is composed of two sums accumulated per byte
8 //     the sum of all bytes, s2 is the sum of all s1 values
9 //     are done modulo 65521. s1 is initialized to 1, s2 to
10 //     Adler-32 checksum is stored as s2*65536 + s1 in most
11 //     significant-byte first (network) order.
12 package adler32
13
14 import "hash"
15
16 const (
17     mod = 65521
18 )
19
20 // The size of an Adler-32 checksum in bytes.
21 const Size = 4
22
23 // digest represents the partial evaluation of a checksum.
24 type digest struct {
25     // invariant: (a < mod && b < mod) || a <= b
26     // invariant: a + b + 255 <= 0xffffffff
27     a, b uint32
28 }
29
30 func (d *digest) Reset() { d.a, d.b = 1, 0 }
31
32 // New returns a new hash.Hash32 computing the Adler-32 chec
33 func New() hash.Hash32 {
34     d := new(digest)
35     d.Reset()
36     return d
37 }
38
39 func (d *digest) Size() int { return Size }
40
41 func (d *digest) BlockSize() int { return 1 }
```

```

42
43 // Add p to the running checksum a, b.
44 func update(a, b uint32, p []byte) (aa, bb uint32) {
45     for _, pi := range p {
46         a += uint32(pi)
47         b += a
48         // invariant: a <= b
49         if b > (0xffffffff-255)/2 {
50             a %= mod
51             b %= mod
52             // invariant: a < mod && b < mod
53         } else {
54             // invariant: a + b + 255 <= 2 * b +
55         }
56     }
57     return a, b
58 }
59
60 // Return the 32-bit checksum corresponding to a, b.
61 func finish(a, b uint32) uint32 {
62     if b >= mod {
63         a %= mod
64         b %= mod
65     }
66     return b<<16 | a
67 }
68
69 func (d *digest) Write(p []byte) (nn int, err error) {
70     d.a, d.b = update(d.a, d.b, p)
71     return len(p), nil
72 }
73
74 func (d *digest) Sum32() uint32 { return finish(d.a, d.b) }
75
76 func (d *digest) Sum(in []byte) []byte {
77     s := d.Sum32()
78     in = append(in, byte(s>>24))
79     in = append(in, byte(s>>16))
80     in = append(in, byte(s>>8))
81     in = append(in, byte(s))
82     return in
83 }
84
85 // Checksum returns the Adler-32 checksum of data.
86 func Checksum(data []byte) uint32 { return finish(update(1,

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/hash/crc32/crc32.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package crc32 implements the 32-bit cyclic redundancy che
6 // checksum. See http://en.wikipedia.org/wiki/Cyclic\_redunda
7 // information.
8 package crc32
9
10 import (
11     "hash"
12     "sync"
13 )
14
15 // The size of a CRC-32 checksum in bytes.
16 const Size = 4
17
18 // Predefined polynomials.
19 const (
20     // Far and away the most common CRC-32 polynomial.
21     // Used by ethernet (IEEE 802.3), v.42, fddi, gzip,
22     IEEE = 0xedb88320
23
24     // Castagnoli's polynomial, used in iSCSI.
25     // Has better error detection characteristics than I
26     // http://dx.doi.org/10.1109/26.231911
27     Castagnoli = 0x82f63b78
28
29     // Koopman's polynomial.
30     // Also has better error detection characteristics t
31     // http://dx.doi.org/10.1109/DSN.2002.1028931
32     Koopman = 0xeb31d82e
33 )
34
35 // Table is a 256-word table representing the polynomial for
36 type Table [256]uint32
37
38 // castagnoliTable points to a lazily initialized Table for
39 // polynomial. MakeTable will always return this value when
40 // Castagnoli table so we can compare against it to find whe
41 // using this polynomial.
```

```

42 var castagnoliTable *Table
43 var castagnoliOnce sync.Once
44
45 func castagnoliInit() {
46     castagnoliTable = makeTable(Castagnoli)
47 }
48
49 // IEEETable is the table for the IEEE polynomial.
50 var IEEETable = makeTable(IEEE)
51
52 // MakeTable returns the Table constructed from the specific
53 func MakeTable(poly uint32) *Table {
54     switch poly {
55     case IEEE:
56         return IEEETable
57     case Castagnoli:
58         castagnoliOnce.Do(castagnoliInit)
59         return castagnoliTable
60     }
61     return makeTable(poly)
62 }
63
64 // makeTable returns the Table constructed from the specific
65 func makeTable(poly uint32) *Table {
66     t := new(Table)
67     for i := 0; i < 256; i++ {
68         crc := uint32(i)
69         for j := 0; j < 8; j++ {
70             if crc&1 == 1 {
71                 crc = (crc >> 1) ^ poly
72             } else {
73                 crc >>= 1
74             }
75         }
76         t[i] = crc
77     }
78     return t
79 }
80
81 // digest represents the partial evaluation of a checksum.
82 type digest struct {
83     crc uint32
84     tab *Table
85 }
86
87 // New creates a new hash.Hash32 computing the CRC-32 checks
88 // using the polynomial represented by the Table.
89 func New(tab *Table) hash.Hash32 { return &digest{0, tab} }
90
91 // NewIEEE creates a new hash.Hash32 computing the CRC-32 ch

```

```

92 // using the IEEE polynomial.
93 func NewIEEE() hash.Hash32 { return New(IEEETable) }
94
95 func (d *digest) Size() int { return Size }
96
97 func (d *digest) BlockSize() int { return 1 }
98
99 func (d *digest) Reset() { d.crc = 0 }
100
101 func update(crc uint32, tab *Table, p []byte) uint32 {
102     crc = ^crc
103     for _, v := range p {
104         crc = tab[byte(crc)^v] ^ (crc >> 8)
105     }
106     return ^crc
107 }
108
109 // Update returns the result of adding the bytes in p to the
110 func Update(crc uint32, tab *Table, p []byte) uint32 {
111     if tab == castagnoliTable {
112         return updateCastagnoli(crc, p)
113     }
114     return update(crc, tab, p)
115 }
116
117 func (d *digest) Write(p []byte) (n int, err error) {
118     d.crc = Update(d.crc, d.tab, p)
119     return len(p), nil
120 }
121
122 func (d *digest) Sum32() uint32 { return d.crc }
123
124 func (d *digest) Sum(in []byte) []byte {
125     s := d.Sum32()
126     in = append(in, byte(s>>24))
127     in = append(in, byte(s>>16))
128     in = append(in, byte(s>>8))
129     in = append(in, byte(s))
130     return in
131 }
132
133 // Checksum returns the CRC-32 checksum of data
134 // using the polynomial represented by the Table.
135 func Checksum(data []byte, tab *Table) uint32 { return Updat
136
137 // ChecksumIEEE returns the CRC-32 checksum of data
138 // using the IEEE polynomial.
139 func ChecksumIEEE(data []byte) uint32 { return update(0, IEE

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/hash/crc32/crc32_amd64.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package crc32
6
7 // This file contains the code to call the SSE 4.2 version of
8 // CRC.
9
10 // haveSSE42 is defined in crc_amd64.s and uses CPUID to test
11 // support.
12 func haveSSE42() bool
13
14 // castagnoliSSE42 is defined in crc_amd64.s and uses the SSE
15 // instruction.
16 func castagnoliSSE42(uint32, []byte) uint32
17
18 var sse42 = haveSSE42()
19
20 func updateCastagnoli(crc uint32, p []byte) uint32 {
21     if sse42 {
22         return castagnoliSSE42(crc, p)
23     }
24     return update(crc, castagnoliTable, p)
25 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/hash/crc64/crc64.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package crc64 implements the 64-bit cyclic redundancy che
6 // checksum. See http://en.wikipedia.org/wiki/Cyclic\_redun
7 // information.
8 package crc64
9
10 import "hash"
11
12 // The size of a CRC-64 checksum in bytes.
13 const Size = 8
14
15 // Predefined polynomials.
16 const (
17     // The ISO polynomial, defined in ISO 3309 and used
18     ISO = 0xD800000000000000
19
20     // The ECMA polynomial, defined in ECMA 182.
21     ECMA = 0xC96C5795D7870F42
22 )
23
24 // Table is a 256-word table representing the polynomial for
25 type Table [256]uint64
26
27 // MakeTable returns the Table constructed from the specific
28 func MakeTable(poly uint64) *Table {
29     t := new(Table)
30     for i := 0; i < 256; i++ {
31         crc := uint64(i)
32         for j := 0; j < 8; j++ {
33             if crc&1 == 1 {
34                 crc = (crc >> 1) ^ poly
35             } else {
36                 crc >>= 1
37             }
38         }
39         t[i] = crc
40     }
41     return t

```

```

42 }
43
44 // digest represents the partial evaluation of a checksum.
45 type digest struct {
46     crc uint64
47     tab *Table
48 }
49
50 // New creates a new hash.Hash64 computing the CRC-64 checks
51 // using the polynomial represented by the Table.
52 func New(tab *Table) hash.Hash64 { return &digest{0, tab} }
53
54 func (d *digest) Size() int { return Size }
55
56 func (d *digest) BlockSize() int { return 1 }
57
58 func (d *digest) Reset() { d.crc = 0 }
59
60 func update(crc uint64, tab *Table, p []byte) uint64 {
61     crc = ^crc
62     for _, v := range p {
63         crc = tab[byte(crc)^v] ^ (crc >> 8)
64     }
65     return ^crc
66 }
67
68 // Update returns the result of adding the bytes in p to the
69 func Update(crc uint64, tab *Table, p []byte) uint64 {
70     return update(crc, tab, p)
71 }
72
73 func (d *digest) Write(p []byte) (n int, err error) {
74     d.crc = update(d.crc, d.tab, p)
75     return len(p), nil
76 }
77
78 func (d *digest) Sum64() uint64 { return d.crc }
79
80 func (d *digest) Sum(in []byte) []byte {
81     s := d.Sum64()
82     in = append(in, byte(s>>56))
83     in = append(in, byte(s>>48))
84     in = append(in, byte(s>>40))
85     in = append(in, byte(s>>32))
86     in = append(in, byte(s>>24))
87     in = append(in, byte(s>>16))
88     in = append(in, byte(s>>8))
89     in = append(in, byte(s))
90     return in
91 }

```

```
92
93 // Checksum returns the CRC-64 checksum of data
94 // using the polynomial represented by the Table.
95 func Checksum(data []byte, tab *Table) uint64 { return updat
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/hash/fnv/fnv.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package fnv implements FNV-1 and FNV-1a, non-cryptographi
6 // created by Glenn Fowler, Landon Curt Noll, and Phong Vo.
7 // See http://isthe.com/chongo/tech/comp/fnv/.
8 package fnv
9
10 import (
11     "hash"
12 )
13
14 type (
15     sum32    uint32
16     sum32a   uint32
17     sum64    uint64
18     sum64a   uint64
19 )
20
21 const (
22     offset32 = 2166136261
23     offset64 = 14695981039346656037
24     prime32  = 16777619
25     prime64  = 1099511628211
26 )
27
28 // New32 returns a new 32-bit FNV-1 hash.Hash.
29 func New32() hash.Hash32 {
30     var s sum32 = offset32
31     return &s
32 }
33
34 // New32a returns a new 32-bit FNV-1a hash.Hash.
35 func New32a() hash.Hash32 {
36     var s sum32a = offset32
37     return &s
38 }
39
40 // New64 returns a new 64-bit FNV-1 hash.Hash.
41 func New64() hash.Hash64 {
42     var s sum64 = offset64
43     return &s
44 }
```

```

45
46 // New64a returns a new 64-bit FNV-1a hash.Hash.
47 func New64a() hash.Hash64 {
48     var s sum64a = offset64
49     return &s
50 }
51
52 func (s *sum32) Reset() { *s = offset32 }
53 func (s *sum32a) Reset() { *s = offset32 }
54 func (s *sum64) Reset() { *s = offset64 }
55 func (s *sum64a) Reset() { *s = offset64 }
56
57 func (s *sum32) Sum32() uint32 { return uint32(*s) }
58 func (s *sum32a) Sum32() uint32 { return uint32(*s) }
59 func (s *sum64) Sum64() uint64 { return uint64(*s) }
60 func (s *sum64a) Sum64() uint64 { return uint64(*s) }
61
62 func (s *sum32) Write(data []byte) (int, error) {
63     hash := *s
64     for _, c := range data {
65         hash *= prime32
66         hash ^= sum32(c)
67     }
68     *s = hash
69     return len(data), nil
70 }
71
72 func (s *sum32a) Write(data []byte) (int, error) {
73     hash := *s
74     for _, c := range data {
75         hash ^= sum32a(c)
76         hash *= prime32
77     }
78     *s = hash
79     return len(data), nil
80 }
81
82 func (s *sum64) Write(data []byte) (int, error) {
83     hash := *s
84     for _, c := range data {
85         hash *= prime64
86         hash ^= sum64(c)
87     }
88     *s = hash
89     return len(data), nil
90 }
91
92 func (s *sum64a) Write(data []byte) (int, error) {
93     hash := *s
94     for _, c := range data {

```

```

95             hash ^= sum64a(c)
96             hash *= prime64
97         }
98         *s = hash
99         return len(data), nil
100    }
101
102    func (s *sum32) Size() int { return 4 }
103    func (s *sum32a) Size() int { return 4 }
104    func (s *sum64) Size() int { return 8 }
105    func (s *sum64a) Size() int { return 8 }
106
107    func (s *sum32) BlockSize() int { return 1 }
108    func (s *sum32a) BlockSize() int { return 1 }
109    func (s *sum64) BlockSize() int { return 1 }
110    func (s *sum64a) BlockSize() int { return 1 }
111
112    func (s *sum32) Sum(in []byte) []byte {
113        v := uint32(*s)
114        in = append(in, byte(v>>24))
115        in = append(in, byte(v>>16))
116        in = append(in, byte(v>>8))
117        in = append(in, byte(v))
118        return in
119    }
120
121    func (s *sum32a) Sum(in []byte) []byte {
122        v := uint32(*s)
123        in = append(in, byte(v>>24))
124        in = append(in, byte(v>>16))
125        in = append(in, byte(v>>8))
126        in = append(in, byte(v))
127        return in
128    }
129
130    func (s *sum64) Sum(in []byte) []byte {
131        v := uint64(*s)
132        in = append(in, byte(v>>56))
133        in = append(in, byte(v>>48))
134        in = append(in, byte(v>>40))
135        in = append(in, byte(v>>32))
136        in = append(in, byte(v>>24))
137        in = append(in, byte(v>>16))
138        in = append(in, byte(v>>8))
139        in = append(in, byte(v))
140        return in
141    }
142
143    func (s *sum64a) Sum(in []byte) []byte {

```

```
144         v := uint64(*s)
145         in = append(in, byte(v>>56))
146         in = append(in, byte(v>>48))
147         in = append(in, byte(v>>40))
148         in = append(in, byte(v>>32))
149         in = append(in, byte(v>>24))
150         in = append(in, byte(v>>16))
151         in = append(in, byte(v>>8))
152         in = append(in, byte(v))
153         return in
154     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/html/entity.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package html
6
7 // All entities that do not end with ';' are 6 or fewer byte
8 const longestEntityWithoutSemicolon = 6
9
10 // entity is a map from HTML entity names to their values. T
11 // http://www.whatwg.org/specs/web-apps/current-work/multipa
12 // lists both "amp" and "amp;" as two separate entries.
13 //
14 // Note that the HTML5 list is larger than the HTML4 list at
15 // http://www.w3.org/TR/html4/sgml/entities.html
16 var entity = map[string]rune{
17     "AElig;": '\U000000C6',
18     "AMP;": '\U00000026',
19     "Acute;": '\U000000C1',
20     "Abreve;": '\U00000102',
21     "Acirc;": '\U000000C2',
22     "Acy;": '\U00000410',
23     "Afr;": '\U0001D504',
24     "Agrave;": '\U000000C0',
25     "Alpha;": '\U00000391',
26     "Amacr;": '\U00000100',
27     "And;": '\U00002A53',
28     "Aogon;": '\U00000104',
29     "Aopf;": '\U0001D538',
30     "ApplyFunction;": '\U00002061',
31     "Aring;": '\U000000C5',
32     "Ascr;": '\U0001D49C',
33     "Assign;": '\U00002254',
34     "Atilde;": '\U000000C3',
35     "Auml;": '\U000000C4',
36     "Backslash;": '\U00002216',
37     "Barv;": '\U00002AE7',
38     "Barwed;": '\U00002306',
39     "Bcy;": '\U00000411',
40     "Because;": '\U00002235',
41     "Bernoullis;": '\U0000212C',
42     "Beta;": '\U00000392',
43     "Bfr;": '\U0001D505',
44     "Bopf;": '\U0001D539',
```

45	"Breve;":	'\U000002D8'
46	"Bscr;":	'\U0000212C'
47	"Bumpeq;":	'\U0000224E'
48	"CHcy;":	'\U00000427'
49	"COPY;":	'\U000000A9'
50	"Cacute;":	'\U00000106'
51	"Cap;":	'\U000022D2'
52	"CapitalDifferentialD;":	'\U00002145'
53	"Cayleys;":	'\U0000212D'
54	"Ccaron;":	'\U0000010C'
55	"Ccedil;":	'\U000000C7'
56	"Ccirc;":	'\U00000108'
57	"Cconint;":	'\U00002230'
58	"Cdot;":	'\U0000010A'
59	"Cedilla;":	'\U000000B8'
60	"CenterDot;":	'\U000000B7'
61	"Cfr;":	'\U0000212D'
62	"Chi;":	'\U000003A7'
63	"CircleDot;":	'\U00002299'
64	"CircleMinus;":	'\U00002296'
65	"CirclePlus;":	'\U00002295'
66	"CircleTimes;":	'\U00002297'
67	"ClockwiseContourIntegral;":	'\U00002232'
68	"CloseCurlyDoubleQuote;":	'\U0000201D'
69	"CloseCurlyQuote;":	'\U00002019'
70	"Colon;":	'\U00002237'
71	"Colone;":	'\U00002A74'
72	"Congruent;":	'\U00002261'
73	"Conint;":	'\U0000222F'
74	"ContourIntegral;":	'\U0000222E'
75	"Copf;":	'\U00002102'
76	"Coproduct;":	'\U00002210'
77	"CounterClockwiseContourIntegral;":	'\U00002233'
78	"Cross;":	'\U00002A2F'
79	"Cscr;":	'\U0001D49E'
80	"Cup;":	'\U000022D3'
81	"CupCap;":	'\U0000224D'
82	"DD;":	'\U00002145'
83	"DDottrahd;":	'\U00002911'
84	"DJcy;":	'\U00000402'
85	"DScy;":	'\U00000405'
86	"DZcy;":	'\U0000040F'
87	"Dagger;":	'\U00002021'
88	"Darr;":	'\U000021A1'
89	"Dashv;":	'\U00002AE4'
90	"Dcaron;":	'\U0000010E'
91	"Dcy;":	'\U00000414'
92	"Del;":	'\U00002207'
93	"Delta;":	'\U00000394'
94	"Dfr;":	'\U0001D507'

95	"DiacriticalAcute;":	'\U000000B4'
96	"DiacriticalDot;":	'\U000002D9'
97	"DiacriticalDoubleAcute;":	'\U000002DD'
98	"DiacriticalGrave;":	'\U00000060'
99	"DiacriticalTilde;":	'\U000002DC'
100	"Diamond;":	'\U0000022C4'
101	"DifferentialD;":	'\U000002146'
102	"Dopf;":	'\U0001D53B'
103	"Dot;":	'\U000000A8'
104	"DotDot;":	'\U0000020DC'
105	"DotEqual;":	'\U000002250'
106	"DoubleContourIntegral;":	'\U00000222F'
107	"DoubleDot;":	'\U000000A8'
108	"DoubleDownArrow;":	'\U0000021D3'
109	"DoubleLeftArrow;":	'\U0000021D0'
110	"DoubleLeftRightArrow;":	'\U0000021D4'
111	"DoubleLeftTee;":	'\U000002AE4'
112	"DoubleLongLeftArrow;":	'\U0000027F8'
113	"DoubleLongLeftRightArrow;":	'\U0000027FA'
114	"DoubleLongRightArrow;":	'\U0000027F9'
115	"DoubleRightArrow;":	'\U0000021D2'
116	"DoubleRightTee;":	'\U0000022A8'
117	"DoubleUpArrow;":	'\U0000021D1'
118	"DoubleUpDownArrow;":	'\U0000021D5'
119	"DoubleVerticalBar;":	'\U000002225'
120	"DownArrow;":	'\U000002193'
121	"DownArrowBar;":	'\U000002913'
122	"DownArrowUpArrow;":	'\U0000021F5'
123	"DownBreve;":	'\U000000311'
124	"DownLeftRightVector;":	'\U000002950'
125	"DownLeftTeeVector;":	'\U00000295E'
126	"DownLeftVector;":	'\U0000021BD'
127	"DownLeftVectorBar;":	'\U000002956'
128	"DownRightTeeVector;":	'\U00000295F'
129	"DownRightVector;":	'\U0000021C1'
130	"DownRightVectorBar;":	'\U000002957'
131	"DownTee;":	'\U0000022A4'
132	"DownTeeArrow;":	'\U0000021A7'
133	"Downarrow;":	'\U0000021D3'
134	"Dscr;":	'\U0001D49F'
135	"Dstrok;":	'\U000000110'
136	"ENG;":	'\U00000014A'
137	"ETH;":	'\U0000000D0'
138	"Eacute;":	'\U0000000C9'
139	"Ecaron;":	'\U00000011A'
140	"Ecirc;":	'\U0000000CA'
141	"Ecy;":	'\U00000042D'
142	"Edot;":	'\U000000116'
143	"Efr;":	'\U0001D508'

144	"Egrave;":	'\U000000C8'
145	"Element;":	'\U00002208'
146	"Emacr;":	'\U00000112'
147	"EmptySmallSquare;":	'\U000025FB'
148	"EmptyVerySmallSquare;":	'\U000025AB'
149	"Eogon;":	'\U00000118'
150	"Eopf;":	'\U0001D53C'
151	"Epsilon;":	'\U00000395'
152	"Equal;":	'\U00002A75'
153	"EqualTilde;":	'\U00002242'
154	"Equilibrium;":	'\U000021CC'
155	"Escr;":	'\U00002130'
156	"Esim;":	'\U00002A73'
157	"Eta;":	'\U00000397'
158	"Euml;":	'\U000000CB'
159	"Exists;":	'\U00002203'
160	"ExponentialE;":	'\U00002147'
161	"Fcy;":	'\U00000424'
162	"Ffr;":	'\U0001D509'
163	"FilledSmallSquare;":	'\U000025FC'
164	"FilledVerySmallSquare;":	'\U000025AA'
165	"Fopf;":	'\U0001D53D'
166	"ForAll;":	'\U00002200'
167	"Fouriertrf;":	'\U00002131'
168	"Fscr;":	'\U00002131'
169	"GJcy;":	'\U00000403'
170	"GT;":	'\U0000003E'
171	"Gamma;":	'\U00000393'
172	"Gammad;":	'\U000003DC'
173	"Gbreve;":	'\U0000011E'
174	"Gcedil;":	'\U00000122'
175	"Gcirc;":	'\U0000011C'
176	"Gcy;":	'\U00000413'
177	"Gdot;":	'\U00000120'
178	"Gfr;":	'\U0001D50A'
179	"Gg;":	'\U000022D9'
180	"Gopf;":	'\U0001D53E'
181	"GreaterEqual;":	'\U00002265'
182	"GreaterEqualLess;":	'\U000022DB'
183	"GreaterFullEqual;":	'\U00002267'
184	"GreaterGreater;":	'\U00002AA2'
185	"GreaterLess;":	'\U00002277'
186	"GreaterSlantEqual;":	'\U00002A7E'
187	"GreaterTilde;":	'\U00002273'
188	"Gscr;":	'\U0001D4A2'
189	"Gt;":	'\U0000226B'
190	"HARDcy;":	'\U0000042A'
191	"Hacek;":	'\U000002C7'
192	"Hat;":	'\U0000005E'

193	"Hcirc;":	'\U00000124'
194	"Hfr;":	'\U0000210C'
195	"HilbertSpace;":	'\U0000210B'
196	"Hopf;":	'\U0000210D'
197	"HorizontalLine;":	'\U00002500'
198	"Hscr;":	'\U0000210B'
199	"Hstrok;":	'\U00000126'
200	"HumpDownHump;":	'\U0000224E'
201	"HumpEqual;":	'\U0000224F'
202	"IEcy;":	'\U00000415'
203	"IJlig;":	'\U00000132'
204	"IOcy;":	'\U00000401'
205	"Iacute;":	'\U000000CD'
206	"Icirc;":	'\U000000CE'
207	"Icy;":	'\U00000418'
208	"Idot;":	'\U00000130'
209	"Ifr;":	'\U00002111'
210	"Igrave;":	'\U000000CC'
211	"Im;":	'\U00002111'
212	"Imacr;":	'\U0000012A'
213	"ImaginaryI;":	'\U00002148'
214	"Implies;":	'\U000021D2'
215	"Int;":	'\U0000222C'
216	"Integral;":	'\U0000222B'
217	"Intersection;":	'\U000022C2'
218	"InvisibleComma;":	'\U00002063'
219	"InvisibleTimes;":	'\U00002062'
220	"Iogon;":	'\U0000012E'
221	"Iopf;":	'\U0001D540'
222	"Iota;":	'\U00000399'
223	"Iscr;":	'\U00002110'
224	"Itilde;":	'\U00000128'
225	"Iukcy;":	'\U00000406'
226	"Iuml;":	'\U000000CF'
227	"Jcirc;":	'\U00000134'
228	"Jcy;":	'\U00000419'
229	"Jfr;":	'\U0001D50D'
230	"Jopf;":	'\U0001D541'
231	"Jscr;":	'\U0001D4A5'
232	"Jsercy;":	'\U00000408'
233	"Jukcy;":	'\U00000404'
234	"KHcy;":	'\U00000425'
235	"KJcy;":	'\U0000040C'
236	"Kappa;":	'\U0000039A'
237	"Kcedil;":	'\U00000136'
238	"Kcy;":	'\U0000041A'
239	"Kfr;":	'\U0001D50E'
240	"Kopf;":	'\U0001D542'
241	"Kscr;":	'\U0001D4A6'
242	"LJcy;":	'\U00000409'

243	"LT;":	'\U0000003C'
244	"Lacute;":	'\U00000139'
245	"Lambda;":	'\U0000039B'
246	"Lang;":	'\U000027EA'
247	"Laplacetrif;":	'\U00002112'
248	"Larr;":	'\U0000219E'
249	"Lcaron;":	'\U0000013D'
250	"Lcedil;":	'\U0000013B'
251	"Lcy;":	'\U0000041B'
252	"LeftAngleBracket;":	'\U000027E8'
253	"LeftArrow;":	'\U00002190'
254	"LeftArrowBar;":	'\U000021E4'
255	"LeftArrowRightArrow;":	'\U000021C6'
256	"LeftCeiling;":	'\U00002308'
257	"LeftDoubleBracket;":	'\U000027E6'
258	"LeftDownTeeVector;":	'\U00002961'
259	"LeftDownVector;":	'\U000021C3'
260	"LeftDownVectorBar;":	'\U00002959'
261	"LeftFloor;":	'\U0000230A'
262	"LeftRightArrow;":	'\U00002194'
263	"LeftRightVector;":	'\U0000294E'
264	"LeftTee;":	'\U000022A3'
265	"LeftTeeArrow;":	'\U000021A4'
266	"LeftTeeVector;":	'\U0000295A'
267	"LeftTriangle;":	'\U000022B2'
268	"LeftTriangleBar;":	'\U000029CF'
269	"LeftTriangleEqual;":	'\U000022B4'
270	"LeftUpDownVector;":	'\U00002951'
271	"LeftUpTeeVector;":	'\U00002960'
272	"LeftUpVector;":	'\U000021BF'
273	"LeftUpVectorBar;":	'\U00002958'
274	"LeftVector;":	'\U000021BC'
275	"LeftVectorBar;":	'\U00002952'
276	"Leftarrow;":	'\U000021D0'
277	"Leftrightarrow;":	'\U000021D4'
278	"LessEqualGreater;":	'\U000022DA'
279	"LessFullEqual;":	'\U00002266'
280	"LessGreater;":	'\U00002276'
281	"LessLess;":	'\U00002AA1'
282	"LessSlantEqual;":	'\U00002A7D'
283	"LessTilde;":	'\U00002272'
284	"Lfr;":	'\U0001D50F'
285	"Ll;":	'\U000022D8'
286	"Lleftarrow;":	'\U000021DA'
287	"Lmidot;":	'\U0000013F'
288	"LongLeftArrow;":	'\U000027F5'
289	"LongLeftRightArrow;":	'\U000027F7'
290	"LongRightArrow;":	'\U000027F6'
291	"Longleftarrow;":	'\U000027F8'

292	"Longleftarrow;":	'\U000027FA'
293	"Longrightarrow;":	'\U000027F9'
294	"Lopf;":	'\U0001D543'
295	"LowerLeftArrow;":	'\U00002199'
296	"LowerRightArrow;":	'\U00002198'
297	"Lscr;":	'\U00002112'
298	"Lsh;":	'\U000021B0'
299	"Lstrok;":	'\U00000141'
300	"Lt;":	'\U0000226A'
301	"Map;":	'\U00002905'
302	"Mcy;":	'\U0000041C'
303	"MediumSpace;":	'\U0000205F'
304	"Mellintrf;":	'\U00002133'
305	"Mfr;":	'\U0001D510'
306	"MinusPlus;":	'\U00002213'
307	"Mopf;":	'\U0001D544'
308	"Mscr;":	'\U00002133'
309	"Mu;":	'\U0000039C'
310	"NJcy;":	'\U0000040A'
311	"Nacute;":	'\U00000143'
312	"Ncaron;":	'\U00000147'
313	"Ncedil;":	'\U00000145'
314	"Ncy;":	'\U0000041D'
315	"NegativeMediumSpace;":	'\U0000200B'
316	"NegativeThickSpace;":	'\U0000200B'
317	"NegativeThinSpace;":	'\U0000200B'
318	"NegativeVeryThinSpace;":	'\U0000200B'
319	"NestedGreaterGreater;":	'\U0000226B'
320	"NestedLessLess;":	'\U0000226A'
321	"NewLine;":	'\U0000000A'
322	"Nfr;":	'\U0001D511'
323	"NoBreak;":	'\U00002060'
324	"NonBreakingSpace;":	'\U000000A0'
325	"Nopf;":	'\U00002115'
326	"Not;":	'\U00002AEC'
327	"NotCongruent;":	'\U00002262'
328	"NotCupCap;":	'\U0000226D'
329	"NotDoubleVerticalBar;":	'\U00002226'
330	"NotElement;":	'\U00002209'
331	"NotEqual;":	'\U00002260'
332	"NotExists;":	'\U00002204'
333	"NotGreater;":	'\U0000226F'
334	"NotGreaterEqual;":	'\U00002271'
335	"NotGreaterLess;":	'\U00002279'
336	"NotGreaterTilde;":	'\U00002275'
337	"NotLeftTriangle;":	'\U000022EA'
338	"NotLeftTriangleEqual;":	'\U000022EC'
339	"NotLess;":	'\U0000226E'
340	"NotLessEqual;":	'\U00002270'

341	"NotLessGreater;":	'\U00002278'
342	"NotLessTilde;":	'\U00002274'
343	"NotPrecedes;":	'\U00002280'
344	"NotPrecedesSlantEqual;":	'\U000022E0'
345	"NotReverseElement;":	'\U0000220C'
346	"NotRightTriangle;":	'\U000022EB'
347	"NotRightTriangleEqual;":	'\U000022ED'
348	"NotSquareSubsetEqual;":	'\U000022E2'
349	"NotSquareSupersetEqual;":	'\U000022E3'
350	"NotSubsetEqual;":	'\U00002288'
351	"NotSucceeds;":	'\U00002281'
352	"NotSucceedsSlantEqual;":	'\U000022E1'
353	"NotSupersetEqual;":	'\U00002289'
354	"NotTilde;":	'\U00002241'
355	"NotTildeEqual;":	'\U00002244'
356	"NotTildeFullEqual;":	'\U00002247'
357	"NotTildeTilde;":	'\U00002249'
358	"NotVerticalBar;":	'\U00002224'
359	"Nscr;":	'\U0001D4A9'
360	"Ntilde;":	'\U000000D1'
361	"Nu;":	'\U0000039D'
362	"OElig;":	'\U00000152'
363	"Oacute;":	'\U000000D3'
364	"Ocirc;":	'\U000000D4'
365	"Ocy;":	'\U0000041E'
366	"Odblac;":	'\U00000150'
367	"Ofr;":	'\U0001D512'
368	"Ograve;":	'\U000000D2'
369	"Omacr;":	'\U0000014C'
370	"Omega;":	'\U000003A9'
371	"Omicron;":	'\U0000039F'
372	"Oopf;":	'\U0001D546'
373	"OpenCurlyDoubleQuote;":	'\U0000201C'
374	"OpenCurlyQuote;":	'\U00002018'
375	"Or;":	'\U00002A54'
376	"Oscr;":	'\U0001D4AA'
377	"Oslash;":	'\U000000D8'
378	"Otilde;":	'\U000000D5'
379	"Otimes;":	'\U00002A37'
380	"Ouml;":	'\U000000D6'
381	"OverBar;":	'\U0000203E'
382	"OverBrace;":	'\U000023DE'
383	"OverBracket;":	'\U000023B4'
384	"OverParenthesis;":	'\U000023DC'
385	"PartialD;":	'\U00002202'
386	"Pcy;":	'\U0000041F'
387	"Pfr;":	'\U0001D513'
388	"Phi;":	'\U000003A6'
389	"Pi;":	'\U000003A0'
390	"PlusMinus;":	'\U000000B1'

391	"Poincareplane;":	'\U0000210C'
392	"Popf;":	'\U00002119'
393	"Pr;":	'\U00002ABB'
394	"Precedes;":	'\U0000227A'
395	"PrecedesEqual;":	'\U00002AAF'
396	"PrecedesSlantEqual;":	'\U0000227C'
397	"PrecedesTilde;":	'\U0000227E'
398	"Prime;":	'\U00002033'
399	"Product;":	'\U0000220F'
400	"Proportion;":	'\U00002237'
401	"Proportional;":	'\U0000221D'
402	"Pscr;":	'\U0001D4AB'
403	"Psi;":	'\U000003A8'
404	"QUOT;":	'\U00000022'
405	"Qfr;":	'\U0001D514'
406	"Qopf;":	'\U0000211A'
407	"Qscr;":	'\U0001D4AC'
408	"RBarr;":	'\U00002910'
409	"REG;":	'\U000000AE'
410	"Racute;":	'\U00000154'
411	"Rang;":	'\U000027EB'
412	"Rarr;":	'\U000021A0'
413	"Rarrtl;":	'\U00002916'
414	"Rcaron;":	'\U00000158'
415	"Rcedil;":	'\U00000156'
416	"Rcy;":	'\U00000420'
417	"Re;":	'\U0000211C'
418	"ReverseElement;":	'\U0000220B'
419	"ReverseEquilibrium;":	'\U000021CB'
420	"ReverseUpEquilibrium;":	'\U0000296F'
421	"Rfr;":	'\U0000211C'
422	"Rho;":	'\U000003A1'
423	"RightAngleBracket;":	'\U000027E9'
424	"RightArrow;":	'\U00002192'
425	"RightArrowBar;":	'\U000021E5'
426	"RightArrowLeftArrow;":	'\U000021C4'
427	"RightCeiling;":	'\U00002309'
428	"RightDoubleBracket;":	'\U000027E7'
429	"RightDownTeeVector;":	'\U0000295D'
430	"RightDownVector;":	'\U000021C2'
431	"RightDownVectorBar;":	'\U00002955'
432	"RightFloor;":	'\U0000230B'
433	"RightTee;":	'\U000022A2'
434	"RightTeeArrow;":	'\U000021A6'
435	"RightTeeVector;":	'\U0000295B'
436	"RightTriangle;":	'\U000022B3'
437	"RightTriangleBar;":	'\U000029D0'
438	"RightTriangleEqual;":	'\U000022B5'
439	"RightUpDownVector;":	'\U0000294F'

440	"RightUpTeeVector;":	'\U0000295C'
441	"RightUpVector;":	'\U000021BE'
442	"RightUpVectorBar;":	'\U00002954'
443	"RightVector;":	'\U000021C0'
444	"RightVectorBar;":	'\U00002953'
445	"Rightarrow;":	'\U000021D2'
446	"Ropf;":	'\U0000211D'
447	"RoundImplies;":	'\U00002970'
448	"Rrightarrow;":	'\U000021DB'
449	"Rscr;":	'\U0000211B'
450	"Rsh;":	'\U000021B1'
451	"RuleDelayed;":	'\U000029F4'
452	"SHCHcy;":	'\U00000429'
453	"SHcy;":	'\U00000428'
454	"SOFTcy;":	'\U0000042C'
455	"Sacute;":	'\U0000015A'
456	"Sc;":	'\U00002ABC'
457	"Scaron;":	'\U00000160'
458	"Scedil;":	'\U0000015E'
459	"Scirc;":	'\U0000015C'
460	"Scy;":	'\U00000421'
461	"Sfr;":	'\U0001D516'
462	"ShortDownArrow;":	'\U00002193'
463	"ShortLeftArrow;":	'\U00002190'
464	"ShortRightArrow;":	'\U00002192'
465	"ShortUpArrow;":	'\U00002191'
466	"Sigma;":	'\U000003A3'
467	"SmallCircle;":	'\U00002218'
468	"Sopf;":	'\U0001D54A'
469	"Sqrt;":	'\U0000221A'
470	"Square;":	'\U000025A1'
471	"SquareIntersection;":	'\U00002293'
472	"SquareSubset;":	'\U0000228F'
473	"SquareSubsetEqual;":	'\U00002291'
474	"SquareSuperset;":	'\U00002290'
475	"SquareSupersetEqual;":	'\U00002292'
476	"SquareUnion;":	'\U00002294'
477	"Sscr;":	'\U0001D4AE'
478	"Star;":	'\U000022C6'
479	"Sub;":	'\U000022D0'
480	"Subset;":	'\U000022D0'
481	"SubsetEqual;":	'\U00002286'
482	"Succeeds;":	'\U0000227B'
483	"SucceedsEqual;":	'\U00002AB0'
484	"SucceedsSlantEqual;":	'\U0000227D'
485	"SucceedsTilde;":	'\U0000227F'
486	"SuchThat;":	'\U0000220B'
487	"Sum;":	'\U00002211'
488	"Sup;":	'\U000022D1'

489	"Superset;":	'\U00002283'
490	"SupersetEqual;":	'\U00002287'
491	"Supset;":	'\U000022D1'
492	"THORN;":	'\U000000DE'
493	"TRADE;":	'\U00002122'
494	"TSHcy;":	'\U0000040B'
495	"TScy;":	'\U00000426'
496	"Tab;":	'\U00000009'
497	"Tau;":	'\U000003A4'
498	"Tcaron;":	'\U00000164'
499	"Tcedil;":	'\U00000162'
500	"Tcy;":	'\U00000422'
501	"Tfr;":	'\U0001D517'
502	"Therefore;":	'\U00002234'
503	"Theta;":	'\U00000398'
504	"ThinSpace;":	'\U00002009'
505	"Tilde;":	'\U0000223C'
506	"TildeEqual;":	'\U00002243'
507	"TildeFullEqual;":	'\U00002245'
508	"TildeTilde;":	'\U00002248'
509	"Topf;":	'\U0001D54B'
510	"TripleDot;":	'\U000020DB'
511	"Tscr;":	'\U0001D4AF'
512	"Tstrok;":	'\U00000166'
513	"Uacute;":	'\U000000DA'
514	"Uarr;":	'\U0000219F'
515	"Uarrocir;":	'\U00002949'
516	"Ubrcy;":	'\U0000040E'
517	"Ubreve;":	'\U0000016C'
518	"Ucirc;":	'\U000000DB'
519	"Ucy;":	'\U00000423'
520	"Udblac;":	'\U00000170'
521	"Ufr;":	'\U0001D518'
522	"Ugrave;":	'\U000000D9'
523	"Umacr;":	'\U0000016A'
524	"UnderBar;":	'\U0000005F'
525	"UnderBrace;":	'\U000023DF'
526	"UnderBracket;":	'\U000023B5'
527	"UnderParenthesis;":	'\U000023DD'
528	"Union;":	'\U000022C3'
529	"UnionPlus;":	'\U0000228E'
530	"Uogon;":	'\U00000172'
531	"Uopf;":	'\U0001D54C'
532	"UpArrow;":	'\U00002191'
533	"UpArrowBar;":	'\U00002912'
534	"UpArrowDownArrow;":	'\U000021C5'
535	"UpDownArrow;":	'\U00002195'
536	"UpEquilibrium;":	'\U0000296E'
537	"UpTee;":	'\U000022A5'
538	"UpTeeArrow;":	'\U000021A5'

539	"Uparrow;":	'\U000021D1'
540	"Updownarrow;":	'\U000021D5'
541	"UpperLeftArrow;":	'\U00002196'
542	"UpperRightArrow;":	'\U00002197'
543	"Upsi;":	'\U000003D2'
544	"Upsilon;":	'\U000003A5'
545	"Uring;":	'\U0000016E'
546	"Uscr;":	'\U0001D4B0'
547	"Utilde;":	'\U00000168'
548	"Uuml;":	'\U000000DC'
549	"VDash;":	'\U000022AB'
550	"Vbar;":	'\U00002AEB'
551	"Vcy;":	'\U00000412'
552	"Vdash;":	'\U000022A9'
553	"Vdashl;":	'\U00002AE6'
554	"Vee;":	'\U000022C1'
555	"Verbar;":	'\U00002016'
556	"Vert;":	'\U00002016'
557	"VerticalBar;":	'\U00002223'
558	"VerticalLine;":	'\U0000007C'
559	"VerticalSeparator;":	'\U00002758'
560	"VerticalTilde;":	'\U00002240'
561	"VeryThinSpace;":	'\U0000200A'
562	"Vfr;":	'\U0001D519'
563	"Vopf;":	'\U0001D54D'
564	"Vscr;":	'\U0001D4B1'
565	"Vvdash;":	'\U000022AA'
566	"Wcirc;":	'\U00000174'
567	"Wedge;":	'\U000022C0'
568	"Wfr;":	'\U0001D51A'
569	"Wopf;":	'\U0001D54E'
570	"Wscr;":	'\U0001D4B2'
571	"Xfr;":	'\U0001D51B'
572	"Xi;":	'\U0000039E'
573	"Xopf;":	'\U0001D54F'
574	"Xscr;":	'\U0001D4B3'
575	"YAcy;":	'\U0000042F'
576	"YIcy;":	'\U00000407'
577	"YUcy;":	'\U0000042E'
578	"Yacute;":	'\U000000DD'
579	"Ycirc;":	'\U00000176'
580	"Ycy;":	'\U0000042B'
581	"Yfr;":	'\U0001D51C'
582	"Yopf;":	'\U0001D550'
583	"Yscr;":	'\U0001D4B4'
584	"Yuml;":	'\U00000178'
585	"ZHcy;":	'\U00000416'
586	"Zacute;":	'\U00000179'
587	"Zcaron;":	'\U0000017D'

588	"Zcy;":	'\U00000417'
589	"Zdot;":	'\U0000017B'
590	"ZeroWidthSpace;":	'\U0000200B'
591	"Zeta;":	'\U00000396'
592	"Zfr;":	'\U00002128'
593	"Zopf;":	'\U00002124'
594	"Zscr;":	'\U0001D4B5'
595	"aacute;":	'\U000000E1'
596	"abreve;":	'\U00000103'
597	"ac;":	'\U0000223E'
598	"acd;":	'\U0000223F'
599	"acirc;":	'\U000000E2'
600	"acute;":	'\U000000B4'
601	"acy;":	'\U00000430'
602	"aelig;":	'\U000000E6'
603	"af;":	'\U00002061'
604	"afr;":	'\U0001D51E'
605	"agrave;":	'\U000000E0'
606	"alefsym;":	'\U00002135'
607	"aleph;":	'\U00002135'
608	"alpha;":	'\U000003B1'
609	"amacr;":	'\U00000101'
610	"amalg;":	'\U00002A3F'
611	"amp;":	'\U00000026'
612	"and;":	'\U00002227'
613	"andand;":	'\U00002A55'
614	"andd;":	'\U00002A5C'
615	"andslope;":	'\U00002A58'
616	"andv;":	'\U00002A5A'
617	"ang;":	'\U00002220'
618	"ange;":	'\U000029A4'
619	"angle;":	'\U00002220'
620	"angmsd;":	'\U00002221'
621	"angmsdaa;":	'\U000029A8'
622	"angmsdab;":	'\U000029A9'
623	"angmsdac;":	'\U000029AA'
624	"angmsdad;":	'\U000029AB'
625	"angmsdae;":	'\U000029AC'
626	"angmsdaf;":	'\U000029AD'
627	"angmsdag;":	'\U000029AE'
628	"angmsdah;":	'\U000029AF'
629	"angrt;":	'\U0000221F'
630	"angrtvb;":	'\U000022BE'
631	"angrtvbd;":	'\U0000299D'
632	"angsph;":	'\U00002222'
633	"angst;":	'\U000000C5'
634	"angzarr;":	'\U0000237C'
635	"aogon;":	'\U00000105'
636	"aopf;":	'\U0001D552'

637	"ap;":	'\U00002248'
638	"apE;":	'\U00002A70'
639	"apacir;":	'\U00002A6F'
640	"ape;":	'\U0000224A'
641	"apid;":	'\U0000224B'
642	"apos;":	'\U00000027'
643	"approx;":	'\U00002248'
644	"approxpeq;":	'\U0000224A'
645	"aring;":	'\U000000E5'
646	"ascr;":	'\U0001D4B6'
647	"ast;":	'\U0000002A'
648	"asyp;":	'\U00002248'
649	"asympeq;":	'\U0000224D'
650	"atilde;":	'\U000000E3'
651	"auml;":	'\U000000E4'
652	"awconint;":	'\U00002233'
653	"awint;":	'\U00002A11'
654	"bNot;":	'\U00002AED'
655	"backcong;":	'\U0000224C'
656	"backepsilon;":	'\U000003F6'
657	"backprime;":	'\U00002035'
658	"backsim;":	'\U0000223D'
659	"backsimeq;":	'\U000022CD'
660	"barvee;":	'\U000022BD'
661	"barwed;":	'\U00002305'
662	"barwedge;":	'\U00002305'
663	"bbrk;":	'\U000023B5'
664	"bbrktbrk;":	'\U000023B6'
665	"bcong;":	'\U0000224C'
666	"bcy;":	'\U00000431'
667	"bdquo;":	'\U0000201E'
668	"becaus;":	'\U00002235'
669	"because;":	'\U00002235'
670	"bemptyv;":	'\U000029B0'
671	"bepsi;":	'\U000003F6'
672	"bernou;":	'\U0000212C'
673	"beta;":	'\U000003B2'
674	"beth;":	'\U00002136'
675	"between;":	'\U0000226C'
676	"bfr;":	'\U0001D51F'
677	"bigcap;":	'\U000022C2'
678	"bigcirc;":	'\U000025EF'
679	"bigcup;":	'\U000022C3'
680	"bigodot;":	'\U00002A00'
681	"bigoplus;":	'\U00002A01'
682	"bigotimes;":	'\U00002A02'
683	"bigsqcup;":	'\U00002A06'
684	"bigstar;":	'\U00002605'
685	"bigtriangledown;":	'\U000025BD'
686	"bigtriangleup;":	'\U000025B3'

687	"biguplus;":	'\U00002A04'
688	"bigvee;":	'\U000022C1'
689	"bigwedge;":	'\U000022C0'
690	"bkarow;":	'\U0000290D'
691	"blacklozenge;":	'\U000029EB'
692	"blacksquare;":	'\U000025AA'
693	"blacktriangle;":	'\U000025B4'
694	"blacktriangledown;":	'\U000025BE'
695	"blacktriangleleft;":	'\U000025C2'
696	"blacktriangleright;":	'\U000025B8'
697	"blank;":	'\U00002423'
698	"blk12;":	'\U00002592'
699	"blk14;":	'\U00002591'
700	"blk34;":	'\U00002593'
701	"block;":	'\U00002588'
702	"bnot;":	'\U00002310'
703	"bopf;":	'\U0001D553'
704	"bot;":	'\U000022A5'
705	"bottom;":	'\U000022A5'
706	"bowtie;":	'\U000022C8'
707	"boxDL;":	'\U00002557'
708	"boxDR;":	'\U00002554'
709	"boxDl;":	'\U00002556'
710	"boxDr;":	'\U00002553'
711	"boxH;":	'\U00002550'
712	"boxHD;":	'\U00002566'
713	"boxHU;":	'\U00002569'
714	"boxHd;":	'\U00002564'
715	"boxHu;":	'\U00002567'
716	"boxUL;":	'\U0000255D'
717	"boxUR;":	'\U0000255A'
718	"boxUl;":	'\U0000255C'
719	"boxUr;":	'\U00002559'
720	"boxV;":	'\U00002551'
721	"boxVH;":	'\U0000256C'
722	"boxVL;":	'\U00002563'
723	"boxVR;":	'\U00002560'
724	"boxVh;":	'\U0000256B'
725	"boxVl;":	'\U00002562'
726	"boxVr;":	'\U0000255F'
727	"boxbox;":	'\U000029C9'
728	"boxdL;":	'\U00002555'
729	"boxdR;":	'\U00002552'
730	"boxdl;":	'\U00002510'
731	"boxdr;":	'\U0000250C'
732	"boxh;":	'\U00002500'
733	"boxhD;":	'\U00002565'
734	"boxhU;":	'\U00002568'
735	"boxhd;":	'\U0000252C'

736	"boxhu;":	'\U00002534'
737	"boxminus;":	'\U0000229F'
738	"boxplus;":	'\U0000229E'
739	"boxtimes;":	'\U000022A0'
740	"boxuL;":	'\U0000255B'
741	"boxuR;":	'\U00002558'
742	"boxuL;":	'\U00002518'
743	"boxur;":	'\U00002514'
744	"boxv;":	'\U00002502'
745	"boxvH;":	'\U0000256A'
746	"boxvL;":	'\U00002561'
747	"boxvR;":	'\U0000255E'
748	"boxvh;":	'\U0000253C'
749	"boxvL;":	'\U00002524'
750	"boxvr;":	'\U0000251C'
751	"bprime;":	'\U00002035'
752	"breve;":	'\U000002D8'
753	"brvbar;":	'\U000000A6'
754	"bscr;":	'\U0001D4B7'
755	"bsemi;":	'\U0000204F'
756	"bsim;":	'\U0000223D'
757	"bsime;":	'\U000022CD'
758	"bsol;":	'\U0000005C'
759	"bsolb;":	'\U000029C5'
760	"bsolhsub;":	'\U000027C8'
761	"bull;":	'\U00002022'
762	"bullet;":	'\U00002022'
763	"bump;":	'\U0000224E'
764	"bumpE;":	'\U00002AAE'
765	"bumpe;":	'\U0000224F'
766	"bumpeq;":	'\U0000224F'
767	"cacute;":	'\U00000107'
768	"cap;":	'\U00002229'
769	"capand;":	'\U00002A44'
770	"capbrcup;":	'\U00002A49'
771	"capcap;":	'\U00002A4B'
772	"capcup;":	'\U00002A47'
773	"capdot;":	'\U00002A40'
774	"caret;":	'\U00002041'
775	"caron;":	'\U000002C7'
776	"ccaps;":	'\U00002A4D'
777	"ccaron;":	'\U0000010D'
778	"ccedil;":	'\U000000E7'
779	"ccirc;":	'\U00000109'
780	"ccups;":	'\U00002A4C'
781	"ccupssm;":	'\U00002A50'
782	"cdot;":	'\U0000010B'
783	"cedil;":	'\U000000B8'
784	"cemptyv;":	'\U000029B2'

785	"cent;":	'\U000000A2'
786	"centerdot;":	'\U000000B7'
787	"cfr;":	'\U0001D520'
788	"chcy;":	'\U00000447'
789	"check;":	'\U00002713'
790	"checkmark;":	'\U00002713'
791	"chi;":	'\U000003C7'
792	"cir;":	'\U000025CB'
793	"cirE;":	'\U000029C3'
794	"circ;":	'\U000002C6'
795	"circeq;":	'\U00002257'
796	"circlearrowleft;":	'\U000021BA'
797	"circlearrowright;":	'\U000021BB'
798	"circledR;":	'\U000000AE'
799	"circledS;":	'\U000024C8'
800	"circledast;":	'\U0000229B'
801	"circledcirc;":	'\U0000229A'
802	"circleddash;":	'\U0000229D'
803	"cire;":	'\U00002257'
804	"cirfnint;":	'\U00002A10'
805	"cirmid;":	'\U00002AEF'
806	"cirscir;":	'\U000029C2'
807	"clubs;":	'\U00002663'
808	"clubsuit;":	'\U00002663'
809	"colon;":	'\U0000003A'
810	"colone;":	'\U00002254'
811	"coloneq;":	'\U00002254'
812	"comma;":	'\U0000002C'
813	"commat;":	'\U00000040'
814	"comp;":	'\U00002201'
815	"compfn;":	'\U00002218'
816	"complement;":	'\U00002201'
817	"complexes;":	'\U00002102'
818	"cong;":	'\U00002245'
819	"congdot;":	'\U00002A6D'
820	"conint;":	'\U0000222E'
821	"copf;":	'\U0001D554'
822	"coprod;":	'\U00002210'
823	"copy;":	'\U000000A9'
824	"copysr;":	'\U00002117'
825	"crarr;":	'\U000021B5'
826	"cross;":	'\U00002717'
827	"cscr;":	'\U0001D4B8'
828	"csub;":	'\U00002ACF'
829	"csube;":	'\U00002AD1'
830	"csup;":	'\U00002AD0'
831	"csupe;":	'\U00002AD2'
832	"ctdot;":	'\U000022EF'
833	"cudarrrl;":	'\U00002938'
834	"cudarrrr;":	'\U00002935'

835	"cuepr;":	'\U000022DE',
836	"cuesc;":	'\U000022DF',
837	"cularr;":	'\U000021B6',
838	"cularrp;":	'\U0000293D',
839	"cup;":	'\U0000222A',
840	"cupbrcap;":	'\U00002A48',
841	"cupcap;":	'\U00002A46',
842	"cupcup;":	'\U00002A4A',
843	"cupdot;":	'\U0000228D',
844	"cupor;":	'\U00002A45',
845	"curarr;":	'\U000021B7',
846	"curarrm;":	'\U0000293C',
847	"curlyeqprec;":	'\U000022DE',
848	"curlyeqsucc;":	'\U000022DF',
849	"curlyvee;":	'\U000022CE',
850	"curlywedge;":	'\U000022CF',
851	"curren;":	'\U000000A4',
852	"curvearrowleft;":	'\U000021B6',
853	"curvearrowright;":	'\U000021B7',
854	"cuvee;":	'\U000022CE',
855	"cuwed;":	'\U000022CF',
856	"cwconint;":	'\U00002232',
857	"cwint;":	'\U00002231',
858	"cylcty;":	'\U0000232D',
859	"dArr;":	'\U000021D3',
860	"dHar;":	'\U00002965',
861	"dagger;":	'\U00002020',
862	"daleth;":	'\U00002138',
863	"darr;":	'\U00002193',
864	"dash;":	'\U00002010',
865	"dashv;":	'\U000022A3',
866	"dbkarow;":	'\U0000290F',
867	"dblac;":	'\U000002DD',
868	"dcaron;":	'\U0000010F',
869	"dcy;":	'\U00000434',
870	"dd;":	'\U00002146',
871	"ddagger;":	'\U00002021',
872	"ddarr;":	'\U000021CA',
873	"ddotseq;":	'\U00002A77',
874	"deg;":	'\U000000B0',
875	"delta;":	'\U000003B4',
876	"emptyv;":	'\U000029B1',
877	"dfisht;":	'\U0000297F',
878	"dfr;":	'\U0001D521',
879	"dharl;":	'\U000021C3',
880	"dharr;":	'\U000021C2',
881	"diam;":	'\U000022C4',
882	"diamond;":	'\U000022C4',
883	"diamondsuit;":	'\U00002666',

884	"diams;":	'\U00002666'
885	"die;":	'\U000000A8'
886	"digamma;":	'\U000003DD'
887	"disin;":	'\U000022F2'
888	"div;":	'\U000000F7'
889	"divide;":	'\U000000F7'
890	"divideontimes;":	'\U000022C7'
891	"divonx;":	'\U000022C7'
892	"djcy;":	'\U00000452'
893	"dlcorn;":	'\U0000231E'
894	"dlcrop;":	'\U0000230D'
895	"dollar;":	'\U00000024'
896	"dopf;":	'\U0001D555'
897	"dot;":	'\U000002D9'
898	"doteq;":	'\U00002250'
899	"doteqdot;":	'\U00002251'
900	"dotminus;":	'\U00002238'
901	"dotplus;":	'\U00002214'
902	"dotsquare;":	'\U000022A1'
903	"doublebarwedge;":	'\U00002306'
904	"downarrow;":	'\U00002193'
905	"downdownarrows;":	'\U000021CA'
906	"downharpoonleft;":	'\U000021C3'
907	"downharpoonright;":	'\U000021C2'
908	"drbkarow;":	'\U00002910'
909	"drcorn;":	'\U0000231F'
910	"drcrop;":	'\U0000230C'
911	"dscr;":	'\U0001D4B9'
912	"dscy;":	'\U00000455'
913	"dsol;":	'\U000029F6'
914	"dstrok;":	'\U00000111'
915	"dtdot;":	'\U000022F1'
916	"dtri;":	'\U000025BF'
917	"dtrif;":	'\U000025BE'
918	"duarr;":	'\U000021F5'
919	"duhar;":	'\U0000296F'
920	"dwangle;":	'\U000029A6'
921	"dzcy;":	'\U0000045F'
922	"dzigarr;":	'\U000027FF'
923	"eDDot;":	'\U00002A77'
924	"eDot;":	'\U00002251'
925	"eacute;":	'\U000000E9'
926	"easter;":	'\U00002A6E'
927	"ecaron;":	'\U0000011B'
928	"ecir;":	'\U00002256'
929	"ecirc;":	'\U000000EA'
930	"ecolon;":	'\U00002255'
931	"ecy;":	'\U0000044D'
932	"edot;":	'\U00000117'

933	"ee;":	'\U00002147'
934	"efDot;":	'\U00002252'
935	"efr;":	'\U0001D522'
936	"eg;":	'\U00002A9A'
937	"egrave;":	'\U000000E8'
938	"egs;":	'\U00002A96'
939	"egsdot;":	'\U00002A98'
940	"el;":	'\U00002A99'
941	"elinters;":	'\U000023E7'
942	"ell;":	'\U00002113'
943	"els;":	'\U00002A95'
944	"elsdot;":	'\U00002A97'
945	"emacr;":	'\U00000113'
946	"empty;":	'\U00002205'
947	"emptyset;":	'\U00002205'
948	"emptyv;":	'\U00002205'
949	"emsp;":	'\U00002003'
950	"emsp13;":	'\U00002004'
951	"emsp14;":	'\U00002005'
952	"eng;":	'\U0000014B'
953	"ensp;":	'\U00002002'
954	"eogon;":	'\U00000119'
955	"eopf;":	'\U0001D556'
956	"epar;":	'\U000022D5'
957	"eparsl;":	'\U000029E3'
958	"eplus;":	'\U00002A71'
959	"epsi;":	'\U000003B5'
960	"epsilon;":	'\U000003B5'
961	"epsiv;":	'\U000003F5'
962	"eqcirc;":	'\U00002256'
963	"eqcolon;":	'\U00002255'
964	"eqsim;":	'\U00002242'
965	"eqslantgtr;":	'\U00002A96'
966	"eqslantless;":	'\U00002A95'
967	"equals;":	'\U0000003D'
968	"equest;":	'\U0000225F'
969	"equiv;":	'\U00002261'
970	"equivDD;":	'\U00002A78'
971	"eqvpar;":	'\U000029E5'
972	"erDot;":	'\U00002253'
973	"erarr;":	'\U00002971'
974	"escr;":	'\U0000212F'
975	"esdot;":	'\U00002250'
976	"esim;":	'\U00002242'
977	"eta;":	'\U000003B7'
978	"eth;":	'\U000000F0'
979	"euml;":	'\U000000EB'
980	"euro;":	'\U000020AC'
981	"excl;":	'\U00000021'
982	"exist;":	'\U00002203'

983	"expectation;":	'\U00002130'
984	"exponentiale;":	'\U00002147'
985	"fallingdotseq;":	'\U00002252'
986	"fcy;":	'\U00000444'
987	"female;":	'\U00002640'
988	"ffilig;":	'\U0000FB03'
989	"fflig;":	'\U0000FB00'
990	"ffllig;":	'\U0000FB04'
991	"ffr;":	'\U0001D523'
992	"filig;":	'\U0000FB01'
993	"flat;":	'\U0000266D'
994	"fllig;":	'\U0000FB02'
995	"fltns;":	'\U000025B1'
996	"fnof;":	'\U00000192'
997	"fopf;":	'\U0001D557'
998	"forall;":	'\U00002200'
999	"fork;":	'\U000022D4'
1000	"forkv;":	'\U00002AD9'
1001	"fpartint;":	'\U00002A0D'
1002	"frac12;":	'\U000000BD'
1003	"frac13;":	'\U00002153'
1004	"frac14;":	'\U000000BC'
1005	"frac15;":	'\U00002155'
1006	"frac16;":	'\U00002159'
1007	"frac18;":	'\U0000215B'
1008	"frac23;":	'\U00002154'
1009	"frac25;":	'\U00002156'
1010	"frac34;":	'\U000000BE'
1011	"frac35;":	'\U00002157'
1012	"frac38;":	'\U0000215C'
1013	"frac45;":	'\U00002158'
1014	"frac56;":	'\U0000215A'
1015	"frac58;":	'\U0000215D'
1016	"frac78;":	'\U0000215E'
1017	"frasl;":	'\U00002044'
1018	"frown;":	'\U00002322'
1019	"fscr;":	'\U0001D4BB'
1020	"gE;":	'\U00002267'
1021	"gEl;":	'\U00002A8C'
1022	"gacute;":	'\U000001F5'
1023	"gamma;":	'\U000003B3'
1024	"gammad;":	'\U000003DD'
1025	"gap;":	'\U00002A86'
1026	"gbreve;":	'\U0000011F'
1027	"gcirc;":	'\U0000011D'
1028	"gcy;":	'\U00000433'
1029	"gdot;":	'\U00000121'
1030	"ge;":	'\U00002265'
1031	"gel;":	'\U000022DB'

1032	"geq;":	'\U00002265'
1033	"geqq;":	'\U00002267'
1034	"geqslant;":	'\U00002A7E'
1035	"ges;":	'\U00002A7E'
1036	"gescc;":	'\U00002AA9'
1037	"gesdot;":	'\U00002A80'
1038	"gesdoto;":	'\U00002A82'
1039	"gesdotol;":	'\U00002A84'
1040	"gesles;":	'\U00002A94'
1041	"gfr;":	'\U0001D524'
1042	"gg;":	'\U0000226B'
1043	"ggg;":	'\U000022D9'
1044	"gimel;":	'\U00002137'
1045	"gjcy;":	'\U00000453'
1046	"gl;":	'\U00002277'
1047	"glE;":	'\U00002A92'
1048	"gla;":	'\U00002AA5'
1049	"glj;":	'\U00002AA4'
1050	"gnE;":	'\U00002269'
1051	"gnap;":	'\U00002A8A'
1052	"gnapprox;":	'\U00002A8A'
1053	"gne;":	'\U00002A88'
1054	"gneq;":	'\U00002A88'
1055	"gneqq;":	'\U00002269'
1056	"gnsim;":	'\U000022E7'
1057	"gopf;":	'\U0001D558'
1058	"grave;":	'\U00000060'
1059	"gscr;":	'\U0000210A'
1060	"gsim;":	'\U00002273'
1061	"gsime;":	'\U00002A8E'
1062	"gsiml;":	'\U00002A90'
1063	"gt;":	'\U0000003E'
1064	"gtcc;":	'\U00002AA7'
1065	"gtcir;":	'\U00002A7A'
1066	"gtdot;":	'\U000022D7'
1067	"gtlPar;":	'\U00002995'
1068	"gtquest;":	'\U00002A7C'
1069	"gtrapprox;":	'\U00002A86'
1070	"gtrarr;":	'\U00002978'
1071	"gtrdot;":	'\U000022D7'
1072	"gtreqless;":	'\U000022DB'
1073	"gtreqqless;":	'\U00002A8C'
1074	"gtrless;":	'\U00002277'
1075	"gtrsim;":	'\U00002273'
1076	"hArr;":	'\U000021D4'
1077	"hairsp;":	'\U0000200A'
1078	"half;":	'\U000000BD'
1079	"hamilt;":	'\U0000210B'
1080	"hardcy;":	'\U0000044A'

1081	"harr;":	'\U00002194'
1082	"harrcir;":	'\U00002948'
1083	"harrw;":	'\U000021AD'
1084	"hbar;":	'\U0000210F'
1085	"hcirc;":	'\U00000125'
1086	"hearts;":	'\U00002665'
1087	"heartsuit;":	'\U00002665'
1088	"hellip;":	'\U00002026'
1089	"hercon;":	'\U000022B9'
1090	"hfr;":	'\U0001D525'
1091	"hksearrow;":	'\U00002925'
1092	"hkswarrow;":	'\U00002926'
1093	"hoarr;":	'\U000021FF'
1094	"homtth;":	'\U0000223B'
1095	"hookleftarrow;":	'\U000021A9'
1096	"hookrightarrow;":	'\U000021AA'
1097	"hopf;":	'\U0001D559'
1098	"horbar;":	'\U00002015'
1099	"hscr;":	'\U0001D4BD'
1100	"hslash;":	'\U0000210F'
1101	"hstrok;":	'\U00000127'
1102	"hybull;":	'\U00002043'
1103	"hyphen;":	'\U00002010'
1104	"iacute;":	'\U000000ED'
1105	"ic;":	'\U00002063'
1106	"icirc;":	'\U000000EE'
1107	"icy;":	'\U00000438'
1108	"iecy;":	'\U00000435'
1109	"iexcl;":	'\U000000A1'
1110	"iff;":	'\U000021D4'
1111	"ifr;":	'\U0001D526'
1112	"igrave;":	'\U000000EC'
1113	"ii;":	'\U00002148'
1114	"iiiint;":	'\U00002A0C'
1115	"iiint;":	'\U0000222D'
1116	"iinfin;":	'\U000029DC'
1117	"iiota;":	'\U00002129'
1118	"ijlig;":	'\U00000133'
1119	"imacr;":	'\U0000012B'
1120	"image;":	'\U00002111'
1121	"imagline;":	'\U00002110'
1122	"imagpart;":	'\U00002111'
1123	"imath;":	'\U00000131'
1124	"imof;":	'\U000022B7'
1125	"imped;":	'\U000001B5'
1126	"in;":	'\U00002208'
1127	"incare;":	'\U00002105'
1128	"infin;":	'\U0000221E'
1129	"infintie;":	'\U000029DD'
1130	"inodot;":	'\U00000131'

1131	"int;":	'\U0000222B'
1132	"intcal;":	'\U000022BA'
1133	"integers;":	'\U00002124'
1134	"intercal;":	'\U000022BA'
1135	"intlarhk;":	'\U00002A17'
1136	"intprod;":	'\U00002A3C'
1137	"iocy;":	'\U00000451'
1138	"iogon;":	'\U0000012F'
1139	"iopf;":	'\U0001D55A'
1140	"iota;":	'\U000003B9'
1141	"iprod;":	'\U00002A3C'
1142	"iquest;":	'\U000000BF'
1143	"iscr;":	'\U0001D4BE'
1144	"isin;":	'\U00002208'
1145	"isinE;":	'\U000022F9'
1146	"isindot;":	'\U000022F5'
1147	"isins;":	'\U000022F4'
1148	"isinsv;":	'\U000022F3'
1149	"isinv;":	'\U00002208'
1150	"it;":	'\U00002062'
1151	"itilde;":	'\U00000129'
1152	"iukcy;":	'\U00000456'
1153	"iuml;":	'\U000000EF'
1154	"jcirc;":	'\U00000135'
1155	"jcy;":	'\U00000439'
1156	"jfr;":	'\U0001D527'
1157	"jmath;":	'\U00000237'
1158	"jopf;":	'\U0001D55B'
1159	"jscr;":	'\U0001D4BF'
1160	"jsercy;":	'\U00000458'
1161	"jukcy;":	'\U00000454'
1162	"kappa;":	'\U000003BA'
1163	"kappav;":	'\U000003F0'
1164	"kcedil;":	'\U00000137'
1165	"kcy;":	'\U0000043A'
1166	"kfr;":	'\U0001D528'
1167	"kgreen;":	'\U00000138'
1168	"khcy;":	'\U00000445'
1169	"kjcy;":	'\U0000045C'
1170	"kopf;":	'\U0001D55C'
1171	"kscr;":	'\U0001D4C0'
1172	"lAarr;":	'\U000021DA'
1173	"lArr;":	'\U000021D0'
1174	"lAtail;":	'\U0000291B'
1175	"lBarr;":	'\U0000290E'
1176	"lE;":	'\U00002266'
1177	"lEg;":	'\U00002A8B'
1178	"lHar;":	'\U00002962'
1179	"lacute;":	'\U0000013A'

1180	"laemptyv;":	'\U000029B4',
1181	"lagran;":	'\U00002112',
1182	"lambda;":	'\U000003BB',
1183	"lang;":	'\U000027E8',
1184	"langd;":	'\U00002991',
1185	"langle;":	'\U000027E8',
1186	"lap;":	'\U00002A85',
1187	"laquo;":	'\U000000AB',
1188	"larr;":	'\U00002190',
1189	"larrb;":	'\U000021E4',
1190	"larrbfs;":	'\U0000291F',
1191	"larrfs;":	'\U0000291D',
1192	"larrhk;":	'\U000021A9',
1193	"larrlp;":	'\U000021AB',
1194	"larrpl;":	'\U00002939',
1195	"larrsim;":	'\U00002973',
1196	"larrtl;":	'\U000021A2',
1197	"lat;":	'\U00002AAB',
1198	"latail;":	'\U00002919',
1199	"late;":	'\U00002AAD',
1200	"lbarr;":	'\U0000290C',
1201	"lbrk;":	'\U00002772',
1202	"lbrace;":	'\U0000007B',
1203	"lbrack;":	'\U0000005B',
1204	"lbrke;":	'\U0000298B',
1205	"lbrksld;":	'\U0000298F',
1206	"lbrkslu;":	'\U0000298D',
1207	"lcaron;":	'\U0000013E',
1208	"lcedil;":	'\U0000013C',
1209	"lceil;":	'\U00002308',
1210	"lcub;":	'\U0000007B',
1211	"lcy;":	'\U0000043B',
1212	"ldca;":	'\U00002936',
1213	"ldquo;":	'\U0000201C',
1214	"ldquor;":	'\U0000201E',
1215	"ldrdhar;":	'\U00002967',
1216	"ldrushar;":	'\U0000294B',
1217	"ldsh;":	'\U000021B2',
1218	"le;":	'\U00002264',
1219	"leftarrow;":	'\U00002190',
1220	"leftarrowtail;":	'\U000021A2',
1221	"leftharpoondown;":	'\U000021BD',
1222	"leftharpoonup;":	'\U000021BC',
1223	"leftleftarrows;":	'\U000021C7',
1224	"leftrightarrow;":	'\U00002194',
1225	"leftrightharpoons;":	'\U000021C6',
1226	"leftrightharpoons;":	'\U000021CB',
1227	"leftrightsquigarrow;":	'\U000021AD',
1228	"leftthreetimes;":	'\U000022CB',

1229	"leg;":	'\U000022DA'
1230	"leq;":	'\U00002264'
1231	"leqq;":	'\U00002266'
1232	"leqslant;":	'\U00002A7D'
1233	"les;":	'\U00002A7D'
1234	"lescc;":	'\U00002AA8'
1235	"lesdot;":	'\U00002A7F'
1236	"lesdoto;":	'\U00002A81'
1237	"lesdotor;":	'\U00002A83'
1238	"lesges;":	'\U00002A93'
1239	"lessapprox;":	'\U00002A85'
1240	"lessdot;":	'\U000022D6'
1241	"lesseqgtr;":	'\U000022DA'
1242	"lesseqqgtr;":	'\U00002A8B'
1243	"lessgtr;":	'\U00002276'
1244	"lesssim;":	'\U00002272'
1245	"lfisht;":	'\U0000297C'
1246	"lfloor;":	'\U0000230A'
1247	"lfr;":	'\U0001D529'
1248	"lg;":	'\U00002276'
1249	"lgE;":	'\U00002A91'
1250	"lhard;":	'\U000021BD'
1251	"lharu;":	'\U000021BC'
1252	"lharul;":	'\U0000296A'
1253	"lhblk;":	'\U00002584'
1254	"ljcy;":	'\U00000459'
1255	"ll;":	'\U0000226A'
1256	"llarr;":	'\U000021C7'
1257	"llcorner;":	'\U0000231E'
1258	"llhard;":	'\U0000296B'
1259	"lltri;":	'\U000025FA'
1260	"lmidot;":	'\U00000140'
1261	"lmoust;":	'\U000023B0'
1262	"lmoustache;":	'\U000023B0'
1263	"lnE;":	'\U00002268'
1264	"lnap;":	'\U00002A89'
1265	"lnapprox;":	'\U00002A89'
1266	"lne;":	'\U00002A87'
1267	"lneq;":	'\U00002A87'
1268	"lneqq;":	'\U00002268'
1269	"lnsim;":	'\U000022E6'
1270	"loang;":	'\U000027EC'
1271	"loarr;":	'\U000021FD'
1272	"lobrk;":	'\U000027E6'
1273	"longleftarrow;":	'\U000027F5'
1274	"longleftrightarrow;":	'\U000027F7'
1275	"longmapsto;":	'\U000027FC'
1276	"longrightarrow;":	'\U000027F6'
1277	"looparrowleft;":	'\U000021AB'
1278	"looparrowright;":	'\U000021AC'

1279	"lopar;":	'\U00002985'
1280	"lopf;":	'\U0001D55D'
1281	"loplus;":	'\U00002A2D'
1282	"lotimes;":	'\U00002A34'
1283	"lowast;":	'\U00002217'
1284	"lowbar;":	'\U0000005F'
1285	"loz;":	'\U000025CA'
1286	"lozenge;":	'\U000025CA'
1287	"lozf;":	'\U000029EB'
1288	"lpar;":	'\U00000028'
1289	"lparlt;":	'\U00002993'
1290	"lrarr;":	'\U000021C6'
1291	"lrcorner;":	'\U0000231F'
1292	"lrhar;":	'\U000021CB'
1293	"lrhard;":	'\U0000296D'
1294	"lrm;":	'\U0000200E'
1295	"lrtri;":	'\U000022BF'
1296	"lsaquo;":	'\U00002039'
1297	"lscr;":	'\U0001D4C1'
1298	"lsh;":	'\U000021B0'
1299	"lsim;":	'\U00002272'
1300	"lsime;":	'\U00002A8D'
1301	"lsimg;":	'\U00002A8F'
1302	"lsqb;":	'\U0000005B'
1303	"lsquo;":	'\U00002018'
1304	"lsquor;":	'\U0000201A'
1305	"lstrok;":	'\U00000142'
1306	"lt;":	'\U0000003C'
1307	"ltcc;":	'\U00002AA6'
1308	"ltcir;":	'\U00002A79'
1309	"ltdot;":	'\U000022D6'
1310	"lthree;":	'\U000022CB'
1311	"ltimes;":	'\U000022C9'
1312	"ltlarr;":	'\U00002976'
1313	"ltquest;":	'\U00002A7B'
1314	"ltrPar;":	'\U00002996'
1315	"ltri;":	'\U000025C3'
1316	"ltrie;":	'\U000022B4'
1317	"ltrif;":	'\U000025C2'
1318	"lurdshar;":	'\U0000294A'
1319	"luruhar;":	'\U00002966'
1320	"mDDot;":	'\U0000223A'
1321	"macr;":	'\U000000AF'
1322	"male;":	'\U00002642'
1323	"malt;":	'\U00002720'
1324	"maltese;":	'\U00002720'
1325	"map;":	'\U000021A6'
1326	"mapsto;":	'\U000021A6'
1327	"mapstodown;":	'\U000021A7'

1328	"mapstoleft;":	'\U000021A4'
1329	"mapstoup;":	'\U000021A5'
1330	"marker;":	'\U000025AE'
1331	"mcomma;":	'\U00002A29'
1332	"mcy;":	'\U0000043C'
1333	"mdash;":	'\U00002014'
1334	"measuredangle;":	'\U00002221'
1335	"mfr;":	'\U0001D52A'
1336	"mho;":	'\U00002127'
1337	"micro;":	'\U000000B5'
1338	"mid;":	'\U00002223'
1339	"midast;":	'\U0000002A'
1340	"midcir;":	'\U00002AF0'
1341	"middot;":	'\U000000B7'
1342	"minus;":	'\U00002212'
1343	"minusb;":	'\U0000229F'
1344	"minusd;":	'\U00002238'
1345	"minusdu;":	'\U00002A2A'
1346	"mlcp;":	'\U00002ADB'
1347	"mlDR;":	'\U00002026'
1348	"mnplus;":	'\U00002213'
1349	"models;":	'\U000022A7'
1350	"mopf;":	'\U0001D55E'
1351	"mp;":	'\U00002213'
1352	"mscr;":	'\U0001D4C2'
1353	"mstpos;":	'\U0000223E'
1354	"mu;":	'\U000003BC'
1355	"multimap;":	'\U000022B8'
1356	"mumap;":	'\U000022B8'
1357	"nLeftarrow;":	'\U000021CD'
1358	"nLeftrightarrow;":	'\U000021CE'
1359	"nrightarrow;":	'\U000021CF'
1360	"nVDash;":	'\U000022AF'
1361	"nVdash;":	'\U000022AE'
1362	"nabla;":	'\U00002207'
1363	"nacute;":	'\U00000144'
1364	"nap;":	'\U00002249'
1365	"napos;":	'\U00000149'
1366	"napprox;":	'\U00002249'
1367	"natur;":	'\U0000266E'
1368	"natural;":	'\U0000266E'
1369	"naturalS;":	'\U00002115'
1370	"nbsp;":	'\U000000A0'
1371	"ncap;":	'\U00002A43'
1372	"ncaron;":	'\U00000148'
1373	"ncedil;":	'\U00000146'
1374	"ncong;":	'\U00002247'
1375	"ncup;":	'\U00002A42'
1376	"ncy;":	'\U0000043D'

1377	"ndash;":	'\U00002013'
1378	"ne;":	'\U00002260'
1379	"neArr;":	'\U000021D7'
1380	"nearhk;":	'\U00002924'
1381	"nearr;":	'\U00002197'
1382	"nearrow;":	'\U00002197'
1383	"nequiv;":	'\U00002262'
1384	"nesear;":	'\U00002928'
1385	"nexist;":	'\U00002204'
1386	"nexists;":	'\U00002204'
1387	"nfr;":	'\U0001D52B'
1388	"nge;":	'\U00002271'
1389	"ngeq;":	'\U00002271'
1390	"ngsim;":	'\U00002275'
1391	"ngt;":	'\U0000226F'
1392	"ngtr;":	'\U0000226F'
1393	"nhArr;":	'\U000021CE'
1394	"nharr;":	'\U000021AE'
1395	"nhpar;":	'\U00002AF2'
1396	"ni;":	'\U0000220B'
1397	"nis;":	'\U000022FC'
1398	"nisd;":	'\U000022FA'
1399	"niv;":	'\U0000220B'
1400	"njcy;":	'\U0000045A'
1401	"nlArr;":	'\U000021CD'
1402	"nlarr;":	'\U0000219A'
1403	"nldr;":	'\U00002025'
1404	"nle;":	'\U00002270'
1405	"nleftarrow;":	'\U0000219A'
1406	"nleftrightarrow;":	'\U000021AE'
1407	"nleq;":	'\U00002270'
1408	"nless;":	'\U0000226E'
1409	"nlsim;":	'\U00002274'
1410	"nlt;":	'\U0000226E'
1411	"nltri;":	'\U000022EA'
1412	"nltrie;":	'\U000022EC'
1413	"nmid;":	'\U00002224'
1414	"nopf;":	'\U0001D55F'
1415	"not;":	'\U000000AC'
1416	"notin;":	'\U00002209'
1417	"notinva;":	'\U00002209'
1418	"notinvb;":	'\U000022F7'
1419	"notinvc;":	'\U000022F6'
1420	"notni;":	'\U0000220C'
1421	"notniva;":	'\U0000220C'
1422	"notnivb;":	'\U000022FE'
1423	"notnivc;":	'\U000022FD'
1424	"npar;":	'\U00002226'
1425	"nparallel;":	'\U00002226'
1426	"npolint;":	'\U00002A14'

1427	"npr;":	'\U00002280'
1428	"nprcue;":	'\U000022E0'
1429	"nprec;":	'\U00002280'
1430	"nrArr;":	'\U000021CF'
1431	"nrarr;":	'\U0000219B'
1432	"nrightharrow;":	'\U0000219B'
1433	"nrtri;":	'\U000022EB'
1434	"nrtrie;":	'\U000022ED'
1435	"nsc;":	'\U00002281'
1436	"nsccue;":	'\U000022E1'
1437	"nscr;":	'\U0001D4C3'
1438	"nshortmid;":	'\U00002224'
1439	"nshortparallel;":	'\U00002226'
1440	"nsim;":	'\U00002241'
1441	"nsime;":	'\U00002244'
1442	"nsimeq;":	'\U00002244'
1443	"nsmid;":	'\U00002224'
1444	"nspar;":	'\U00002226'
1445	"nsqsube;":	'\U000022E2'
1446	"nsqsupe;":	'\U000022E3'
1447	"nsub;":	'\U00002284'
1448	"nsube;":	'\U00002288'
1449	"nsubseteq;":	'\U00002288'
1450	"nsucc;":	'\U00002281'
1451	"nsup;":	'\U00002285'
1452	"nsupe;":	'\U00002289'
1453	"nsupseteq;":	'\U00002289'
1454	"ntgl;":	'\U00002279'
1455	"ntilde;":	'\U000000F1'
1456	"ntlg;":	'\U00002278'
1457	"ntriangleleft;":	'\U000022EA'
1458	"ntrianglelefteq;":	'\U000022EC'
1459	"ntriangleright;":	'\U000022EB'
1460	"ntrianglerighteq;":	'\U000022ED'
1461	"nu;":	'\U000003BD'
1462	"num;":	'\U00000023'
1463	"numero;":	'\U00002116'
1464	"numsp;":	'\U00002007'
1465	"nvDash;":	'\U000022AD'
1466	"nvHarr;":	'\U00002904'
1467	"nvdash;":	'\U000022AC'
1468	"nvinfin;":	'\U000029DE'
1469	"nvlArr;":	'\U00002902'
1470	"nvrArr;":	'\U00002903'
1471	"nwArr;":	'\U000021D6'
1472	"nwarhk;":	'\U00002923'
1473	"nwarr;":	'\U00002196'
1474	"nwarrow;":	'\U00002196'
1475	"nwnear;":	'\U00002927'

1476	"oS;":	'\U000024C8'
1477	"oacute;":	'\U000000F3'
1478	"oast;":	'\U0000229B'
1479	"ocir;":	'\U0000229A'
1480	"ocirc;":	'\U000000F4'
1481	"ocy;":	'\U0000043E'
1482	"odash;":	'\U0000229D'
1483	"odblac;":	'\U00000151'
1484	"odiv;":	'\U00002A38'
1485	"odot;":	'\U00002299'
1486	"odsold;":	'\U000029BC'
1487	"oelig;":	'\U00000153'
1488	"ofcir;":	'\U000029BF'
1489	"ofr;":	'\U0001D52C'
1490	"ogon;":	'\U000002DB'
1491	"ograve;":	'\U000000F2'
1492	"ogt;":	'\U000029C1'
1493	"ohbar;":	'\U000029B5'
1494	"ohm;":	'\U000003A9'
1495	"oint;":	'\U0000222E'
1496	"olarr;":	'\U000021BA'
1497	"olcir;":	'\U000029BE'
1498	"olcross;":	'\U000029BB'
1499	"oline;":	'\U0000203E'
1500	"olt;":	'\U000029C0'
1501	"omacr;":	'\U0000014D'
1502	"omega;":	'\U000003C9'
1503	"omicron;":	'\U000003BF'
1504	"omid;":	'\U000029B6'
1505	"ominus;":	'\U00002296'
1506	"oopf;":	'\U0001D560'
1507	"opar;":	'\U000029B7'
1508	"operp;":	'\U000029B9'
1509	"oplus;":	'\U00002295'
1510	"or;":	'\U00002228'
1511	"orarr;":	'\U000021BB'
1512	"ord;":	'\U00002A5D'
1513	"order;":	'\U00002134'
1514	"orderof;":	'\U00002134'
1515	"ordf;":	'\U000000AA'
1516	"ordm;":	'\U000000BA'
1517	"origof;":	'\U000022B6'
1518	"oror;":	'\U00002A56'
1519	"orslope;":	'\U00002A57'
1520	"orv;":	'\U00002A5B'
1521	"oscr;":	'\U00002134'
1522	"oslash;":	'\U000000F8'
1523	"osol;":	'\U00002298'
1524	"otilde;":	'\U000000F5'

1525	"otimes;":	'\U00002297'
1526	"otimesas;":	'\U00002A36'
1527	"ouml;":	'\U000000F6'
1528	"ovbar;":	'\U0000233D'
1529	"par;":	'\U00002225'
1530	"para;":	'\U000000B6'
1531	"parallel;":	'\U00002225'
1532	"parsim;":	'\U00002AF3'
1533	"parsl;":	'\U00002AFD'
1534	"part;":	'\U00002202'
1535	"pcy;":	'\U0000043F'
1536	"percent;":	'\U00000025'
1537	"period;":	'\U0000002E'
1538	"permil;":	'\U00002030'
1539	"perp;":	'\U000022A5'
1540	"pertenk;":	'\U00002031'
1541	"pfr;":	'\U0001D52D'
1542	"phi;":	'\U000003C6'
1543	"phiv;":	'\U000003D5'
1544	"phmmat;":	'\U00002133'
1545	"phone;":	'\U0000260E'
1546	"pi;":	'\U000003C0'
1547	"pitchfork;":	'\U000022D4'
1548	"piv;":	'\U000003D6'
1549	"planck;":	'\U0000210F'
1550	"planckh;":	'\U0000210E'
1551	"plankv;":	'\U0000210F'
1552	"plus;":	'\U0000002B'
1553	"plusacir;":	'\U00002A23'
1554	"plusb;":	'\U0000229E'
1555	"pluscir;":	'\U00002A22'
1556	"plusdo;":	'\U00002214'
1557	"plusdu;":	'\U00002A25'
1558	"pluse;":	'\U00002A72'
1559	"plusmn;":	'\U000000B1'
1560	"plussim;":	'\U00002A26'
1561	"plustwo;":	'\U00002A27'
1562	"pm;":	'\U000000B1'
1563	"pointint;":	'\U00002A15'
1564	"popf;":	'\U0001D561'
1565	"pound;":	'\U000000A3'
1566	"pr;":	'\U0000227A'
1567	"prE;":	'\U00002AB3'
1568	"prap;":	'\U00002AB7'
1569	"prcue;":	'\U0000227C'
1570	"pre;":	'\U00002AAF'
1571	"prec;":	'\U0000227A'
1572	"precapprox;":	'\U00002AB7'
1573	"preccurlyeq;":	'\U0000227C'
1574	"preceq;":	'\U00002AAF'

1575	"precnapprox;":	'\U00002AB9'
1576	"precneqq;":	'\U00002AB5'
1577	"precnsim;":	'\U000022E8'
1578	"precsim;":	'\U0000227E'
1579	"prime;":	'\U00002032'
1580	"primes;":	'\U00002119'
1581	"prnE;":	'\U00002AB5'
1582	"prnap;":	'\U00002AB9'
1583	"prnsim;":	'\U000022E8'
1584	"prod;":	'\U0000220F'
1585	"profalar;":	'\U0000232E'
1586	"proflinE;":	'\U00002312'
1587	"profsurf;":	'\U00002313'
1588	"prop;":	'\U0000221D'
1589	"propto;":	'\U0000221D'
1590	"prsim;":	'\U0000227E'
1591	"prurel;":	'\U000022B0'
1592	"pscr;":	'\U0001D4C5'
1593	"psi;":	'\U000003C8'
1594	"puncsp;":	'\U00002008'
1595	"qfr;":	'\U0001D52E'
1596	"qint;":	'\U00002A0C'
1597	"qopf;":	'\U0001D562'
1598	"qprime;":	'\U00002057'
1599	"qscr;":	'\U0001D4C6'
1600	"quaternions;":	'\U0000210D'
1601	"quatint;":	'\U00002A16'
1602	"quest;":	'\U0000003F'
1603	"questeq;":	'\U0000225F'
1604	"quot;":	'\U00000022'
1605	"rAarr;":	'\U000021DB'
1606	"rArr;":	'\U000021D2'
1607	"rAtail;":	'\U0000291C'
1608	"rBarr;":	'\U0000290F'
1609	"rHar;":	'\U00002964'
1610	"racute;":	'\U00000155'
1611	"radic;":	'\U0000221A'
1612	"raemptyv;":	'\U000029B3'
1613	"rang;":	'\U000027E9'
1614	"rangd;":	'\U00002992'
1615	"range;":	'\U000029A5'
1616	"rangle;":	'\U000027E9'
1617	"raquo;":	'\U000000BB'
1618	"rarr;":	'\U00002192'
1619	"rarrap;":	'\U00002975'
1620	"rarrb;":	'\U000021E5'
1621	"rarrbfs;":	'\U00002920'
1622	"rarrc;":	'\U00002933'
1623	"rarrfs;":	'\U0000291E'

1624	"rarrhk;":	'\U000021AA',
1625	"rarrlp;":	'\U000021AC',
1626	"rarrpl;":	'\U00002945',
1627	"rarrsim;":	'\U00002974',
1628	"rarrtl;":	'\U000021A3',
1629	"rarrw;":	'\U0000219D',
1630	"ratail;":	'\U0000291A',
1631	"ratio;":	'\U00002236',
1632	"rationals;":	'\U0000211A',
1633	"rbarr;":	'\U0000290D',
1634	"rbbrk;":	'\U00002773',
1635	"rbrace;":	'\U0000007D',
1636	"rbrack;":	'\U0000005D',
1637	"rbrke;":	'\U0000298C',
1638	"rbrksld;":	'\U0000298E',
1639	"rbrkslu;":	'\U00002990',
1640	"rcaron;":	'\U00000159',
1641	"rcedil;":	'\U00000157',
1642	"rceil;":	'\U00002309',
1643	"rcub;":	'\U0000007D',
1644	"rcy;":	'\U00000440',
1645	"rdca;":	'\U00002937',
1646	"rdldhar;":	'\U00002969',
1647	"rdquo;":	'\U0000201D',
1648	"rdquor;":	'\U0000201D',
1649	"rdsh;":	'\U000021B3',
1650	"real;":	'\U0000211C',
1651	"realine;":	'\U0000211B',
1652	"realpart;":	'\U0000211C',
1653	"reals;":	'\U0000211D',
1654	"rect;":	'\U000025AD',
1655	"reg;":	'\U000000AE',
1656	"rfisht;":	'\U0000297D',
1657	"rfloor;":	'\U0000230B',
1658	"rfr;":	'\U0001D52F',
1659	"rhard;":	'\U000021C1',
1660	"rharu;":	'\U000021C0',
1661	"rharul;":	'\U0000296C',
1662	"rho;":	'\U000003C1',
1663	"rhov;":	'\U000003F1',
1664	"rightarrow;":	'\U00002192',
1665	"rightarrowtail;":	'\U000021A3',
1666	"rightharpoondown;":	'\U000021C1',
1667	"rightharpoonup;":	'\U000021C0',
1668	"rightleftarrows;":	'\U000021C4',
1669	"rightleftharpoons;":	'\U000021CC',
1670	"righttrightarrows;":	'\U000021C9',
1671	"rightsquigarrow;":	'\U0000219D',
1672	"rightthreetimes;":	'\U000022CC',

1673	"ring;":	'\U000002DA'
1674	"risingdotseq;":	'\U00002253'
1675	"rlarr;":	'\U000021C4'
1676	"rlhar;":	'\U000021CC'
1677	"rlm;":	'\U0000200F'
1678	"rmoust;":	'\U000023B1'
1679	"rmoustache;":	'\U000023B1'
1680	"rnmid;":	'\U00002AEE'
1681	"roang;":	'\U000027ED'
1682	"roarr;":	'\U000021FE'
1683	"robrk;":	'\U000027E7'
1684	"ropar;":	'\U00002986'
1685	"ropf;":	'\U0001D563'
1686	"roplus;":	'\U00002A2E'
1687	"rotimes;":	'\U00002A35'
1688	"rpar;":	'\U00000029'
1689	"rpargt;":	'\U00002994'
1690	"rppolint;":	'\U00002A12'
1691	"rrarr;":	'\U000021C9'
1692	"rsaquo;":	'\U0000203A'
1693	"rscr;":	'\U0001D4C7'
1694	"rsh;":	'\U000021B1'
1695	"rsqb;":	'\U0000005D'
1696	"rsquo;":	'\U00002019'
1697	"rsquor;":	'\U00002019'
1698	"rthree;":	'\U000022CC'
1699	"rtimes;":	'\U000022CA'
1700	"rtri;":	'\U000025B9'
1701	"rtrie;":	'\U000022B5'
1702	"rtrif;":	'\U000025B8'
1703	"rtriltri;":	'\U000029CE'
1704	"ruluhar;":	'\U00002968'
1705	"rx;":	'\U0000211E'
1706	"sacute;":	'\U0000015B'
1707	"sbquo;":	'\U0000201A'
1708	"sc;":	'\U0000227B'
1709	"scE;":	'\U00002AB4'
1710	"scap;":	'\U00002AB8'
1711	"scaron;":	'\U00000161'
1712	"sccue;":	'\U0000227D'
1713	"sce;":	'\U00002AB0'
1714	"scedil;":	'\U0000015F'
1715	"scirc;":	'\U0000015D'
1716	"scnE;":	'\U00002AB6'
1717	"scnap;":	'\U00002ABA'
1718	"scnsim;":	'\U000022E9'
1719	"scpolint;":	'\U00002A13'
1720	"scsim;":	'\U0000227F'
1721	"scy;":	'\U00000441'
1722	"sdot;":	'\U000022C5'

1723	"sdotb;":	'\U000022A1'
1724	"sdote;":	'\U00002A66'
1725	"seArr;":	'\U000021D8'
1726	"searhk;":	'\U00002925'
1727	"searr;":	'\U00002198'
1728	"searrow;":	'\U00002198'
1729	"sect;":	'\U000000A7'
1730	"semi;":	'\U0000003B'
1731	"seswar;":	'\U00002929'
1732	"setminus;":	'\U00002216'
1733	"setmn;":	'\U00002216'
1734	"sext;":	'\U00002736'
1735	"sfr;":	'\U0001D530'
1736	"sfrown;":	'\U00002322'
1737	"sharp;":	'\U0000266F'
1738	"shchcy;":	'\U00000449'
1739	"shcy;":	'\U00000448'
1740	"shortmid;":	'\U00002223'
1741	"shortparallel;":	'\U00002225'
1742	"shy;":	'\U000000AD'
1743	"sigma;":	'\U000003C3'
1744	"sigmaf;":	'\U000003C2'
1745	"sigmav;":	'\U000003C2'
1746	"sim;":	'\U0000223C'
1747	"simdot;":	'\U00002A6A'
1748	"sime;":	'\U00002243'
1749	"simeq;":	'\U00002243'
1750	"simg;":	'\U00002A9E'
1751	"simgE;":	'\U00002AA0'
1752	"siml;":	'\U00002A9D'
1753	"simlE;":	'\U00002A9F'
1754	"simne;":	'\U00002246'
1755	"simplus;":	'\U00002A24'
1756	"simrarr;":	'\U00002972'
1757	"slarr;":	'\U00002190'
1758	"smallsetminus;":	'\U00002216'
1759	"smashp;":	'\U00002A33'
1760	"smeparsl;":	'\U000029E4'
1761	"smid;":	'\U00002223'
1762	"smile;":	'\U00002323'
1763	"smt;":	'\U00002AAA'
1764	"smte;":	'\U00002AAC'
1765	"softcy;":	'\U0000044C'
1766	"sol;":	'\U0000002F'
1767	"solb;":	'\U000029C4'
1768	"solbar;":	'\U0000233F'
1769	"sopf;":	'\U0001D564'
1770	"spades;":	'\U00002660'
1771	"spadesuit;":	'\U00002660'

1772	"spar;":	'\U00002225'
1773	"sqcap;":	'\U00002293'
1774	"sqcup;":	'\U00002294'
1775	"sqsub;":	'\U0000228F'
1776	"sqsube;":	'\U00002291'
1777	"sqsubset;":	'\U0000228F'
1778	"sqsubseteq;":	'\U00002291'
1779	"sqsup;":	'\U00002290'
1780	"sqsupe;":	'\U00002292'
1781	"sqsupset;":	'\U00002290'
1782	"sqsupseteq;":	'\U00002292'
1783	"squ;":	'\U000025A1'
1784	"square;":	'\U000025A1'
1785	"squarf;":	'\U000025AA'
1786	"squf;":	'\U000025AA'
1787	"srarr;":	'\U00002192'
1788	"sscr;":	'\U0001D4C8'
1789	"ssetmn;":	'\U00002216'
1790	"ssmile;":	'\U00002323'
1791	"sstarf;":	'\U000022C6'
1792	"star;":	'\U00002606'
1793	"starf;":	'\U00002605'
1794	"straightepsilon;":	'\U000003F5'
1795	"straightphi;":	'\U000003D5'
1796	"strns;":	'\U000000AF'
1797	"sub;":	'\U00002282'
1798	"subE;":	'\U00002AC5'
1799	"subdot;":	'\U00002ABD'
1800	"sube;":	'\U00002286'
1801	"subedot;":	'\U00002AC3'
1802	"submult;":	'\U00002AC1'
1803	"subnE;":	'\U00002ACB'
1804	"subne;":	'\U0000228A'
1805	"subplus;":	'\U00002ABF'
1806	"subrarr;":	'\U00002979'
1807	"subset;":	'\U00002282'
1808	"subseteq;":	'\U00002286'
1809	"subseteqq;":	'\U00002AC5'
1810	"subsetneq;":	'\U0000228A'
1811	"subsetneqq;":	'\U00002ACB'
1812	"subsim;":	'\U00002AC7'
1813	"subsub;":	'\U00002AD5'
1814	"subsup;":	'\U00002AD3'
1815	"succ;":	'\U0000227B'
1816	"succapprox;":	'\U00002AB8'
1817	"succcurlyeq;":	'\U0000227D'
1818	"succeq;":	'\U00002AB0'
1819	"succnapprox;":	'\U00002ABA'
1820	"succneqq;":	'\U00002AB6'

1821	"succnsim;":	'\U000022E9'
1822	"succsim;":	'\U0000227F'
1823	"sum;":	'\U00002211'
1824	"sung;":	'\U0000266A'
1825	"sup;":	'\U00002283'
1826	"sup1;":	'\U000000B9'
1827	"sup2;":	'\U000000B2'
1828	"sup3;":	'\U000000B3'
1829	"supE;":	'\U00002AC6'
1830	"supdot;":	'\U00002ABE'
1831	"supdsub;":	'\U00002AD8'
1832	"supe;":	'\U00002287'
1833	"supedot;":	'\U00002AC4'
1834	"suphsol;":	'\U000027C9'
1835	"suphsub;":	'\U00002AD7'
1836	"suplarr;":	'\U0000297B'
1837	"supmult;":	'\U00002AC2'
1838	"supnE;":	'\U00002ACC'
1839	"supne;":	'\U0000228B'
1840	"supplus;":	'\U00002AC0'
1841	"supset;":	'\U00002283'
1842	"supseteq;":	'\U00002287'
1843	"supseteqq;":	'\U00002AC6'
1844	"supsetneq;":	'\U0000228B'
1845	"supsetneqq;":	'\U00002ACC'
1846	"supsim;":	'\U00002AC8'
1847	"supsub;":	'\U00002AD4'
1848	"supsup;":	'\U00002AD6'
1849	"swArr;":	'\U000021D9'
1850	"swarhk;":	'\U00002926'
1851	"swarr;":	'\U00002199'
1852	"swarrow;":	'\U00002199'
1853	"swnwar;":	'\U0000292A'
1854	"szlig;":	'\U000000DF'
1855	"target;":	'\U00002316'
1856	"tau;":	'\U000003C4'
1857	"tbrk;":	'\U000023B4'
1858	"tcaron;":	'\U00000165'
1859	"tcedil;":	'\U00000163'
1860	"tcy;":	'\U00000442'
1861	"tdot;":	'\U000020DB'
1862	"telrec;":	'\U00002315'
1863	"tfr;":	'\U0001D531'
1864	"there4;":	'\U00002234'
1865	"therefore;":	'\U00002234'
1866	"theta;":	'\U000003B8'
1867	"thetasym;":	'\U000003D1'
1868	"thetav;":	'\U000003D1'
1869	"thickapprox;":	'\U00002248'
1870	"thicksim;":	'\U0000223C'

1871	"thinsp;":	'\U00002009'
1872	"thkap;":	'\U00002248'
1873	"thksim;":	'\U0000223C'
1874	"thorn;":	'\U000000FE'
1875	"tilde;":	'\U000002DC'
1876	"times;":	'\U000000D7'
1877	"timesb;":	'\U000022A0'
1878	"timesbar;":	'\U00002A31'
1879	"timesd;":	'\U00002A30'
1880	"tint;":	'\U0000222D'
1881	"toea;":	'\U00002928'
1882	"top;":	'\U000022A4'
1883	"topbot;":	'\U00002336'
1884	"topcir;":	'\U00002AF1'
1885	"topf;":	'\U0001D565'
1886	"topfork;":	'\U00002ADA'
1887	"tosa;":	'\U00002929'
1888	"tprime;":	'\U00002034'
1889	"trade;":	'\U00002122'
1890	"triangle;":	'\U000025B5'
1891	"triangledown;":	'\U000025BF'
1892	"triangleleft;":	'\U000025C3'
1893	"trianglelefteq;":	'\U000022B4'
1894	"triangleq;":	'\U0000225C'
1895	"triangleright;":	'\U000025B9'
1896	"trianglerighteq;":	'\U000022B5'
1897	"tridot;":	'\U000025EC'
1898	"trie;":	'\U0000225C'
1899	"triminus;":	'\U00002A3A'
1900	"triplus;":	'\U00002A39'
1901	"trisb;":	'\U000029CD'
1902	"tritime;":	'\U00002A3B'
1903	"trpezium;":	'\U000023E2'
1904	"tscr;":	'\U0001D4C9'
1905	"tscy;":	'\U00000446'
1906	"tshcy;":	'\U0000045B'
1907	"tstrok;":	'\U00000167'
1908	"twixt;":	'\U0000226C'
1909	"twoheadleftarrow;":	'\U0000219E'
1910	"twoheadrightarrow;":	'\U000021A0'
1911	"uArr;":	'\U000021D1'
1912	"uHar;":	'\U00002963'
1913	"uacute;":	'\U000000FA'
1914	"uarr;":	'\U00002191'
1915	"ubrcy;":	'\U0000045E'
1916	"ubreve;":	'\U0000016D'
1917	"ucirc;":	'\U000000FB'
1918	"ucy;":	'\U00000443'
1919	"udarr;":	'\U000021C5'

1920	"udblac;":	'\U00000171'
1921	"udhar;":	'\U0000296E'
1922	"ufisht;":	'\U0000297E'
1923	"ufr;":	'\U0001D532'
1924	"ugrave;":	'\U000000F9'
1925	"uharl;":	'\U000021BF'
1926	"uharr;":	'\U000021BE'
1927	"uhblk;":	'\U00002580'
1928	"ulcorn;":	'\U0000231C'
1929	"ulcorner;":	'\U0000231C'
1930	"ulcrop;":	'\U0000230F'
1931	"ultri;":	'\U000025F8'
1932	"umacr;":	'\U0000016B'
1933	"uml;":	'\U000000A8'
1934	"uogon;":	'\U00000173'
1935	"uopf;":	'\U0001D566'
1936	"uparrow;":	'\U00002191'
1937	"updownarrow;":	'\U00002195'
1938	"upharpoonleft;":	'\U000021BF'
1939	"upharpoonright;":	'\U000021BE'
1940	"uplus;":	'\U0000228E'
1941	"upsi;":	'\U000003C5'
1942	"upsih;":	'\U000003D2'
1943	"upsilon;":	'\U000003C5'
1944	"upuparrows;":	'\U000021C8'
1945	"urcorn;":	'\U0000231D'
1946	"urcorner;":	'\U0000231D'
1947	"urcrop;":	'\U0000230E'
1948	"uring;":	'\U0000016F'
1949	"urtri;":	'\U000025F9'
1950	"uscr;":	'\U0001D4CA'
1951	"utdot;":	'\U000022F0'
1952	"utilde;":	'\U00000169'
1953	"utri;":	'\U000025B5'
1954	"utrif;":	'\U000025B4'
1955	"uuarr;":	'\U000021C8'
1956	"uuml;":	'\U000000FC'
1957	"uwangle;":	'\U000029A7'
1958	"vArr;":	'\U000021D5'
1959	"vBar;":	'\U00002AE8'
1960	"vBarv;":	'\U00002AE9'
1961	"vDash;":	'\U000022A8'
1962	"vangrt;":	'\U0000299C'
1963	"varepsilon;":	'\U000003F5'
1964	"varkappa;":	'\U000003F0'
1965	"varnothing;":	'\U00002205'
1966	"varphi;":	'\U000003D5'
1967	"varpi;":	'\U000003D6'
1968	"varpropto;":	'\U0000221D'

1969	"varr;":	'\U00002195'
1970	"varrho;":	'\U000003F1'
1971	"varsigma;":	'\U000003C2'
1972	"vartheta;":	'\U000003D1'
1973	"vartriangleleft;":	'\U000022B2'
1974	"vartriangleright;":	'\U000022B3'
1975	"vcy;":	'\U00000432'
1976	"vdash;":	'\U000022A2'
1977	"vee;":	'\U00002228'
1978	"veebar;":	'\U000022BB'
1979	"veeeq;":	'\U0000225A'
1980	"vellip;":	'\U000022EE'
1981	"verbar;":	'\U0000007C'
1982	"vert;":	'\U0000007C'
1983	"vfr;":	'\U0001D533'
1984	"vltri;":	'\U000022B2'
1985	"vopf;":	'\U0001D567'
1986	"vprop;":	'\U0000221D'
1987	"vrtri;":	'\U000022B3'
1988	"vscr;":	'\U0001D4CB'
1989	"vzigzag;":	'\U0000299A'
1990	"wcirc;":	'\U00000175'
1991	"wedbar;":	'\U00002A5F'
1992	"wedge;":	'\U00002227'
1993	"wedgeq;":	'\U00002259'
1994	"weierp;":	'\U00002118'
1995	"wfr;":	'\U0001D534'
1996	"wopf;":	'\U0001D568'
1997	"wp;":	'\U00002118'
1998	"wr;":	'\U00002240'
1999	"wreath;":	'\U00002240'
2000	"wscr;":	'\U0001D4CC'
2001	"xcap;":	'\U000022C2'
2002	"xcirc;":	'\U000025EF'
2003	"xcup;":	'\U000022C3'
2004	"xdtri;":	'\U000025BD'
2005	"xfr;":	'\U0001D535'
2006	"xhArr;":	'\U000027FA'
2007	"xharr;":	'\U000027F7'
2008	"xi;":	'\U000003BE'
2009	"xlArr;":	'\U000027F8'
2010	"xlarr;":	'\U000027F5'
2011	"xmap;":	'\U000027FC'
2012	"xnis;":	'\U000022FB'
2013	"xodot;":	'\U00002A00'
2014	"xopf;":	'\U0001D569'
2015	"xoplus;":	'\U00002A01'
2016	"xotime;":	'\U00002A02'
2017	"xrArr;":	'\U000027F9'
2018	"xrarr;":	'\U000027F6'

2019	"xscr;":	'\U0001D4CD'
2020	"xscup;":	'\U00002A06'
2021	"xuplus;":	'\U00002A04'
2022	"xutri;":	'\U000025B3'
2023	"xvee;":	'\U000022C1'
2024	"xwedge;":	'\U000022C0'
2025	"yacute;":	'\U000000FD'
2026	"yacy;":	'\U0000044F'
2027	"ycirc;":	'\U00000177'
2028	"ycy;":	'\U0000044B'
2029	"yen;":	'\U000000A5'
2030	"yfr;":	'\U0001D536'
2031	"yicy;":	'\U00000457'
2032	"yopf;":	'\U0001D56A'
2033	"yscr;":	'\U0001D4CE'
2034	"yucy;":	'\U0000044E'
2035	"yuml;":	'\U000000FF'
2036	"zacute;":	'\U0000017A'
2037	"zcaron;":	'\U0000017E'
2038	"zcy;":	'\U00000437'
2039	"zdot;":	'\U0000017C'
2040	"zeetrf;":	'\U00002128'
2041	"zeta;":	'\U000003B6'
2042	"zfr;":	'\U0001D537'
2043	"zhcy;":	'\U00000436'
2044	"zigrarr;":	'\U000021DD'
2045	"zopf;":	'\U0001D56B'
2046	"zscr;":	'\U0001D4CF'
2047	"zwj;":	'\U0000200D'
2048	"zwnj;":	'\U0000200C'
2049	"AElig":	'\U000000C6'
2050	"AMP":	'\U00000026'
2051	"Aacute":	'\U000000C1'
2052	"Acirc":	'\U000000C2'
2053	"Agrave":	'\U000000C0'
2054	"Aring":	'\U000000C5'
2055	"Atilde":	'\U000000C3'
2056	"Auml":	'\U000000C4'
2057	"COPY":	'\U000000A9'
2058	"Ccedil":	'\U000000C7'
2059	"ETH":	'\U000000D0'
2060	"Eacute":	'\U000000C9'
2061	"Ecirc":	'\U000000CA'
2062	"Egrave":	'\U000000C8'
2063	"Euml":	'\U000000CB'
2064	"GT":	'\U0000003E'
2065	"Iacute":	'\U000000CD'
2066	"Icirc":	'\U000000CE'
2067	"Igrave":	'\U000000CC'

2068	"Iuml":	'\U000000CF'
2069	"LT":	'\U0000003C'
2070	"Ntilde":	'\U000000D1'
2071	"Oacute":	'\U000000D3'
2072	"Ocirc":	'\U000000D4'
2073	"Ograve":	'\U000000D2'
2074	"Oslash":	'\U000000D8'
2075	"Otilde":	'\U000000D5'
2076	"Ouml":	'\U000000D6'
2077	"QUOT":	'\U00000022'
2078	"REG":	'\U000000AE'
2079	"THORN":	'\U000000DE'
2080	"Uacute":	'\U000000DA'
2081	"Ucirc":	'\U000000DB'
2082	"Ugrave":	'\U000000D9'
2083	"Uuml":	'\U000000DC'
2084	"Yacute":	'\U000000DD'
2085	"aacute":	'\U000000E1'
2086	"acirc":	'\U000000E2'
2087	"acute":	'\U000000B4'
2088	"aelig":	'\U000000E6'
2089	"agrave":	'\U000000E0'
2090	"amp":	'\U00000026'
2091	"aring":	'\U000000E5'
2092	"atilde":	'\U000000E3'
2093	"auml":	'\U000000E4'
2094	"brvbar":	'\U000000A6'
2095	"ccedil":	'\U000000E7'
2096	"cedil":	'\U000000B8'
2097	"cent":	'\U000000A2'
2098	"copy":	'\U000000A9'
2099	"curren":	'\U000000A4'
2100	"deg":	'\U000000B0'
2101	"divide":	'\U000000F7'
2102	"eacute":	'\U000000E9'
2103	"ecirc":	'\U000000EA'
2104	"egrave":	'\U000000E8'
2105	"eth":	'\U000000F0'
2106	"euml":	'\U000000EB'
2107	"frac12":	'\U000000BD'
2108	"frac14":	'\U000000BC'
2109	"frac34":	'\U000000BE'
2110	"gt":	'\U0000003E'
2111	"iacute":	'\U000000ED'
2112	"icirc":	'\U000000EE'
2113	"iexcl":	'\U000000A1'
2114	"igrave":	'\U000000EC'
2115	"iquest":	'\U000000BF'
2116	"iuml":	'\U000000EF'

```

2117     "laquo":      '\U000000AB',
2118     "lt":        '\U0000003C',
2119     "macr":      '\U000000AF',
2120     "micro":     '\U000000B5',
2121     "middot":    '\U000000B7',
2122     "nbsp":      '\U000000A0',
2123     "not":       '\U000000AC',
2124     "ntilde":    '\U000000F1',
2125     "oacute":    '\U000000F3',
2126     "ocirc":     '\U000000F4',
2127     "ograve":    '\U000000F2',
2128     "ordf":      '\U000000AA',
2129     "ordm":      '\U000000BA',
2130     "oslash":    '\U000000F8',
2131     "otilde":    '\U000000F5',
2132     "ouml":      '\U000000F6',
2133     "para":      '\U000000B6',
2134     "plusmn":    '\U000000B1',
2135     "pound":     '\U000000A3',
2136     "quot":      '\U00000022',
2137     "raquo":     '\U000000BB',
2138     "reg":       '\U000000AE',
2139     "sect":      '\U000000A7',
2140     "shy":       '\U000000AD',
2141     "sup1":      '\U000000B9',
2142     "sup2":      '\U000000B2',
2143     "sup3":      '\U000000B3',
2144     "szlig":     '\U000000DF',
2145     "thorn":     '\U000000FE',
2146     "times":     '\U000000D7',
2147     "uacute":    '\U000000FA',
2148     "ucirc":     '\U000000FB',
2149     "ugrave":    '\U000000F9',
2150     "uml":       '\U000000A8',
2151     "uuml":      '\U000000FC',
2152     "yacute":    '\U000000FD',
2153     "yen":       '\U000000A5',
2154     "yuml":      '\U000000FF',
2155 }
2156
2157 // HTML entities that are two unicode codepoints.
2158 var entity2 = map[string][2]rune{
2159     // TODO(nigeltao): Handle replacements that are wide
2160     // "nLt;":      {'\u226A', '\u20D2'},
2161     // "nGt;":      {'\u226B', '\u20D2'},
2162     "NotEqualTilde;": {'\u2242', '\u0338'},
2163     "NotGreaterFullEqual;": {'\u2267', '\u0338'},
2164     "NotGreaterGreater;": {'\u226B', '\u0338'},
2165     "NotGreaterSlantEqual;": {'\u2A7E', '\u0338'},
2166     "NotHumpDownHump;": {'\u224E', '\u0338'},

```

2167	"NotHumpEqual;":	{'\u224F', '\u0338'},
2168	"NotLeftTriangleBar;":	{'\u29CF', '\u0338'},
2169	"NotLessLess;":	{'\u226A', '\u0338'},
2170	"NotLessSlantEqual;":	{'\u2A7D', '\u0338'},
2171	"NotNestedGreaterGreater;":	{'\u2AA2', '\u0338'},
2172	"NotNestedLessLess;":	{'\u2AA1', '\u0338'},
2173	"NotPrecedesEqual;":	{'\u2AAF', '\u0338'},
2174	"NotRightTriangleBar;":	{'\u29D0', '\u0338'},
2175	"NotSquareSubset;":	{'\u228F', '\u0338'},
2176	"NotSquareSuperset;":	{'\u2290', '\u0338'},
2177	"NotSubset;":	{'\u2282', '\u20D2'},
2178	"NotSucceedsEqual;":	{'\u2AB0', '\u0338'},
2179	"NotSucceedsTilde;":	{'\u227F', '\u0338'},
2180	"NotSuperset;":	{'\u2283', '\u20D2'},
2181	"ThickSpace;":	{'\u205F', '\u200A'},
2182	"acE;":	{'\u223E', '\u0333'},
2183	"bne;":	{'\u003D', '\u20E5'},
2184	"bnequiv;":	{'\u2261', '\u20E5'},
2185	"caps;":	{'\u2229', '\uFE00'},
2186	"cups;":	{'\u222A', '\uFE00'},
2187	"fjlig;":	{'\u0066', '\u006A'},
2188	"gesl;":	{'\u22DB', '\uFE00'},
2189	"gvertneqq;":	{'\u2269', '\uFE00'},
2190	"gvnE;":	{'\u2269', '\uFE00'},
2191	"lates;":	{'\u2AAD', '\uFE00'},
2192	"lesg;":	{'\u22DA', '\uFE00'},
2193	"lvertneqq;":	{'\u2268', '\uFE00'},
2194	"lvnE;":	{'\u2268', '\uFE00'},
2195	"nGg;":	{'\u22D9', '\u0338'},
2196	"nGtv;":	{'\u226B', '\u0338'},
2197	"nLl;":	{'\u22D8', '\u0338'},
2198	"nLtv;":	{'\u226A', '\u0338'},
2199	"nang;":	{'\u2220', '\u20D2'},
2200	"napE;":	{'\u2A70', '\u0338'},
2201	"napid;":	{'\u224B', '\u0338'},
2202	"nbump;":	{'\u224E', '\u0338'},
2203	"nbumpe;":	{'\u224F', '\u0338'},
2204	"ncongdot;":	{'\u2A6D', '\u0338'},
2205	"nedot;":	{'\u2250', '\u0338'},
2206	"nesim;":	{'\u2242', '\u0338'},
2207	"ngE;":	{'\u2267', '\u0338'},
2208	"ngeqq;":	{'\u2267', '\u0338'},
2209	"ngeqslant;":	{'\u2A7E', '\u0338'},
2210	"nges;":	{'\u2A7E', '\u0338'},
2211	"nlE;":	{'\u2266', '\u0338'},
2212	"nleqq;":	{'\u2266', '\u0338'},
2213	"nleqslant;":	{'\u2A7D', '\u0338'},
2214	"nles;":	{'\u2A7D', '\u0338'},
2215	"notinE;":	{'\u22F9', '\u0338'},

```

2216     "notindot;":      {'\u22F5', '\u0338'},
2217     "nparsl;":        {'\u2AFD', '\u20E5'},
2218     "npart;":         {'\u2202', '\u0338'},
2219     "npre;":          {'\u2AAF', '\u0338'},
2220     "npreceq;":       {'\u2AAF', '\u0338'},
2221     "nrarrc;":        {'\u2933', '\u0338'},
2222     "nrarrw;":        {'\u219D', '\u0338'},
2223     "nsce;":          {'\u2AB0', '\u0338'},
2224     "nsubE;":         {'\u2AC5', '\u0338'},
2225     "nsubset;":       {'\u2282', '\u20D2'},
2226     "nsubseteqq;":    {'\u2AC5', '\u0338'},
2227     "nsucceq;":       {'\u2AB0', '\u0338'},
2228     "nsupE;":         {'\u2AC6', '\u0338'},
2229     "nsupset;":       {'\u2283', '\u20D2'},
2230     "nsupseteqq;":    {'\u2AC6', '\u0338'},
2231     "nvap;":          {'\u224D', '\u20D2'},
2232     "nvge;":          {'\u2265', '\u20D2'},
2233     "nvgt;":          {'\u003E', '\u20D2'},
2234     "nvle;":          {'\u2264', '\u20D2'},
2235     "nvlt;":          {'\u003C', '\u20D2'},
2236     "nvltrie;":       {'\u22B4', '\u20D2'},
2237     "nvrtrie;":       {'\u22B5', '\u20D2'},
2238     "nvsim;":         {'\u223C', '\u20D2'},
2239     "race;":          {'\u223D', '\u0331'},
2240     "smtes;":         {'\u2AAC', '\uFE00'},
2241     "sqcaps;":        {'\u2293', '\uFE00'},
2242     "sqcups;":        {'\u2294', '\uFE00'},
2243     "varsubsetneq;":  {'\u228A', '\uFE00'},
2244     "varsubsetneqq;": {'\u2ACB', '\uFE00'},
2245     "varsupsetneq;":  {'\u228B', '\uFE00'},
2246     "varsupsetneqq;": {'\u2ACC', '\uFE00'},
2247     "vnsub;":         {'\u2282', '\u20D2'},
2248     "vnsub;":         {'\u2283', '\u20D2'},
2249     "vsubnE;":        {'\u2ACB', '\uFE00'},
2250     "vsubne;":        {'\u228A', '\uFE00'},
2251     "vsupnE;":        {'\u2ACC', '\uFE00'},
2252     "vsupne;":        {'\u228B', '\uFE00'},
2253 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/html/escape.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package html provides functions for escaping and unescapi
6 package html
7
8 import (
9     "bytes"
10    "strings"
11    "unicode/utf8"
12 )
13
14 type writer interface {
15     WriteString(string) (int, error)
16 }
17
18 // These replacements permit compatibility with old numeric
19 // assumed Windows-1252 encoding.
20 // http://www.whatwg.org/specs/web-apps/current-work/multipa
21 var replacementTable = [...]rune{
22     '\u20AC', // First entry is what 0x80 should be repl
23     '\u0081',
24     '\u201A',
25     '\u0192',
26     '\u201E',
27     '\u2026',
28     '\u2020',
29     '\u2021',
30     '\u02C6',
31     '\u2030',
32     '\u0160',
33     '\u2039',
34     '\u0152',
35     '\u008D',
36     '\u017D',
37     '\u008F',
38     '\u0090',
39     '\u2018',
40     '\u2019',
41     '\u201C',
42     '\u201D',
43     '\u2022',
44     '\u2013',
```

```

45     '\u2014',
46     '\u02DC',
47     '\u2122',
48     '\u0161',
49     '\u203A',
50     '\u0153',
51     '\u009D',
52     '\u017E',
53     '\u0178', // Last entry is 0x9F.
54     // 0x00->' \uFFFFD' is handled programmatically.
55     // 0x0D->' \u000D' is a no-op.
56 }
57
58 // unescapeEntity reads an entity like "&lt;" from b[src:] a
59 // corresponding "<" to b[dst:], returning the incremented d
60 // Precondition: b[src] == '&' && dst <= src.
61 // attribute should be true if parsing an attribute value.
62 func unescapeEntity(b []byte, dst, src int, attribute bool)
63     // http://www.whatwg.org/specs/web-apps/current-work
64
65     // i starts at 1 because we already know that s[0] =
66     // i, s := 1, b[src:]
67
68     if len(s) <= 1 {
69         b[dst] = b[src]
70         return dst + 1, src + 1
71     }
72
73     if s[i] == '#' {
74         if len(s) <= 3 { // We need to have at least
75             b[dst] = b[src]
76             return dst + 1, src + 1
77         }
78         i++
79         c := s[i]
80         hex := false
81         if c == 'x' || c == 'X' {
82             hex = true
83             i++
84         }
85
86         x := '\x00'
87         for i < len(s) {
88             c = s[i]
89             i++
90             if hex {
91                 if '0' <= c && c <= '9' {
92                     x = 16*x + rune(c) -
93                     continue
94                 } else if 'a' <= c && c <= '

```

```

95             x = 16*x + rune(c) -
96             continue
97         } else if 'A' <= c && c <= '
98             x = 16*x + rune(c) -
99             continue
100     }
101     } else if '0' <= c && c <= '9' {
102         x = 10*x + rune(c) - '0'
103         continue
104     }
105     if c != ';' {
106         i--
107     }
108     break
109 }
110
111 if i <= 3 { // No characters matched.
112     b[dst] = b[src]
113     return dst + 1, src + 1
114 }
115
116 if 0x80 <= x && x <= 0x9F {
117     // Replace characters from Windows-1
118     x = replacementTable[x-0x80]
119 } else if x == 0 || (0xD800 <= x && x <= 0xD
120     // Replace invalid characters with t
121     x = '\uFFFD'
122 }
123
124     return dst + utf8.EncodeRune(b[dst:], x), sr
125 }
126
127 // Consume the maximum number of characters possible
128 // consumed characters matching one of the named ref
129
130 for i < len(s) {
131     c := s[i]
132     i++
133     // Lower-cased characters are more common in
134     if 'a' <= c && c <= 'z' || 'A' <= c && c <=
135         continue
136     }
137     if c != ';' {
138         i--
139     }
140     break
141 }
142
143 entityName := string(s[1:i])

```

```

144     if entityName == "" {
145         // No-op.
146     } else if attribute && entityName[len(entityName)-1]
147         // No-op.
148     } else if x := entity[entityName]; x != 0 {
149         return dst + utf8.EncodeRune(b[dst:], x), sr
150     } else if x := entity2[entityName]; x[0] != 0 {
151         dst1 := dst + utf8.EncodeRune(b[dst:], x[0])
152         return dst1 + utf8.EncodeRune(b[dst1:], x[1])
153     } else if !attribute {
154         maxlen := len(entityName) - 1
155         if maxlen > longestEntityWithoutSemicolon {
156             maxlen = longestEntityWithoutSemicolon
157         }
158         for j := maxlen; j > 1; j-- {
159             if x := entity[entityName[:j]]; x !=
160                 return dst + utf8.EncodeRune
161         }
162     }
163 }
164
165     dst1, src1 = dst+i, src+i
166     copy(b[dst:dst1], b[src:src1])
167     return dst1, src1
168 }
169
170 // unescape unescapes b's entities in-place, so that "a&lt;b
171 func unescape(b []byte) []byte {
172     for i, c := range b {
173         if c == '&' {
174             dst, src := unescapeEntity(b, i, i,
175                 for src < len(b) {
176                     c := b[src]
177                     if c == '&' {
178                         dst, src = unescapeE
179                     } else {
180                         b[dst] = c
181                         dst, src = dst+1, sr
182                     }
183                 }
184             return b[0:dst]
185         }
186     }
187     return b
188 }
189
190 // lower lower-cases the A-Z bytes in b in-place, so that "a
191 func lower(b []byte) []byte {
192     for i, c := range b {

```

```

193         if 'A' <= c && c <= 'Z' {
194             b[i] = c + 'a' - 'A'
195         }
196     }
197     return b
198 }
199
200 const escapedChars = `&'<>"`
201
202 func escape(w writer, s string) error {
203     i := strings.IndexAny(s, escapedChars)
204     for i != -1 {
205         if _, err := w.WriteString(s[:i]); err != nil
206             return err
207     }
208     var esc string
209     switch s[i] {
210     case '&':
211         esc = "&"
212     case '\':
213         // "'" is shorter than "&apos;"
214         esc = "'"
215     case '<':
216         esc = "<"
217     case '>':
218         esc = ">"
219     case '"':
220         // """ is shorter than "&quot;".
221         esc = """
222     default:
223         panic("unrecognized escape character")
224     }
225     s = s[i+1:]
226     if _, err := w.WriteString(esc); err != nil
227         return err
228     }
229     i = strings.IndexAny(s, escapedChars)
230 }
231 _, err := w.WriteString(s)
232 return err
233 }
234
235 // EscapeString escapes special characters like "<" to becom
236 // escapes only five such characters: <, >, &, ' and ".
237 // UnescapeString(EscapeString(s)) == s always holds, but th
238 // always true.
239 func EscapeString(s string) string {
240     if strings.IndexAny(s, escapedChars) == -1 {
241         return s
242     }

```

```

243         var buf bytes.Buffer
244         escape(&buf, s)
245         return buf.String()
246     }
247
248     // UnescapeString unescapes entities like "&lt;" to become "
249     // larger range of entities than EscapeString escapes. For e
250     // unescapes to "á", as does "&#225;" and "&xE1;".
251     // UnescapeString(EscapeString(s)) == s always holds, but th
252     // always true.
253     func UnescapeString(s string) string {
254         for _, c := range s {
255             if c == '&' {
256                 return string(unescape([]byte(s)))
257             }
258         }
259         return s
260     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/attr.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "strings"
9 )
10
11 // attrTypeMap[n] describes the value of the given attribute
12 // If an attribute affects (or can mask) the encoding or int
13 // other content, or affects the contents, idempotency, or c
14 // network message, then the value in this map is contentType
15 // This map is derived from HTML5, specifically
16 // http://www.w3.org/TR/html5/Overview.html#attributes-1
17 // as well as "%URI"-typed attributes from
18 // http://www.w3.org/TR/html4/index/attributes.html
19 var attrTypeMap = map[string]contentType{
20     "accept":           contentTypePlain,
21     "accept-charset":  contentTypeUnsafe,
22     "action":          contentTypeURL,
23     "alt":              contentTypePlain,
24     "archive":         contentTypeURL,
25     "async":           contentTypeUnsafe,
26     "autocomplete":    contentTypePlain,
27     "autofocus":       contentTypePlain,
28     "autoplay":        contentTypePlain,
29     "background":      contentTypeURL,
30     "border":          contentTypePlain,
31     "checked":         contentTypePlain,
32     "cite":            contentTypeURL,
33     "challenge":       contentTypeUnsafe,
34     "charset":         contentTypeUnsafe,
35     "class":           contentTypePlain,
36     "classid":         contentTypeURL,
37     "codebase":        contentTypeURL,
38     "cols":            contentTypePlain,
39     "colspan":         contentTypePlain,
40     "content":         contentTypeUnsafe,
41     "contenteditable": contentTypePlain,
```

42	"contextmenu":	contentTypePlain,
43	"controls":	contentTypePlain,
44	"coords":	contentTypePlain,
45	"crossorigin":	contentTypeUnsafe,
46	"data":	contentTypeURL,
47	"datetime":	contentTypePlain,
48	"default":	contentTypePlain,
49	"defer":	contentTypeUnsafe,
50	"dir":	contentTypePlain,
51	"dirname":	contentTypePlain,
52	"disabled":	contentTypePlain,
53	"draggable":	contentTypePlain,
54	"dropzone":	contentTypePlain,
55	"enctype":	contentTypeUnsafe,
56	"for":	contentTypePlain,
57	"form":	contentTypeUnsafe,
58	"formaction":	contentTypeURL,
59	"formenctype":	contentTypeUnsafe,
60	"formmethod":	contentTypeUnsafe,
61	"formnovalidate":	contentTypeUnsafe,
62	"formtarget":	contentTypePlain,
63	"headers":	contentTypePlain,
64	"height":	contentTypePlain,
65	"hidden":	contentTypePlain,
66	"high":	contentTypePlain,
67	"href":	contentTypeURL,
68	"hreflang":	contentTypePlain,
69	"http-equiv":	contentTypeUnsafe,
70	"icon":	contentTypeURL,
71	"id":	contentTypePlain,
72	"ismap":	contentTypePlain,
73	"keytype":	contentTypeUnsafe,
74	"kind":	contentTypePlain,
75	"label":	contentTypePlain,
76	"lang":	contentTypePlain,
77	"language":	contentTypeUnsafe,
78	"list":	contentTypePlain,
79	"longdesc":	contentTypeURL,
80	"loop":	contentTypePlain,
81	"low":	contentTypePlain,
82	"manifest":	contentTypeURL,
83	"max":	contentTypePlain,
84	"maxlength":	contentTypePlain,
85	"media":	contentTypePlain,
86	"mediagroup":	contentTypePlain,
87	"method":	contentTypeUnsafe,
88	"min":	contentTypePlain,
89	"multiple":	contentTypePlain,
90	"name":	contentTypePlain,
91	"novalidate":	contentTypeUnsafe,

```

92         // Skip handler names from
93         // http://www.w3.org/TR/html5/Overview.html#event-ha
94         // since we have special handling in attrType.
95         "open":          contentTypePlain,
96         "optimum":       contentTypePlain,
97         "pattern":       contentTypeUnsafe,
98         "placeholder":   contentTypePlain,
99         "poster":        contentTypeURL,
100        "profile":        contentTypeURL,
101        "preload":        contentTypePlain,
102        "pubdate":        contentTypePlain,
103        "radiogroup":     contentTypePlain,
104        "readonly":       contentTypePlain,
105        "rel":            contentTypeUnsafe,
106        "required":       contentTypePlain,
107        "reversed":       contentTypePlain,
108        "rows":           contentTypePlain,
109        "rowspan":        contentTypePlain,
110        "sandbox":        contentTypeUnsafe,
111        "spellcheck":     contentTypePlain,
112        "scope":          contentTypePlain,
113        "scoped":         contentTypePlain,
114        "seamless":      contentTypePlain,
115        "selected":       contentTypePlain,
116        "shape":          contentTypePlain,
117        "size":           contentTypePlain,
118        "sizes":          contentTypePlain,
119        "span":           contentTypePlain,
120        "src":            contentTypeURL,
121        "srcdoc":         contentTypeHTML,
122        "srclang":        contentTypePlain,
123        "start":          contentTypePlain,
124        "step":           contentTypePlain,
125        "style":          contentTypeCSS,
126        "tabindex":       contentTypePlain,
127        "target":         contentTypePlain,
128        "title":          contentTypePlain,
129        "type":           contentTypeUnsafe,
130        "usemap":         contentTypeURL,
131        "value":          contentTypeUnsafe,
132        "width":          contentTypePlain,
133        "wrap":           contentTypePlain,
134        "xmlns":          contentTypeURL,
135    }
136
137    // attrType returns a conservative (upper-bound on authority
138    // type of the named attribute.
139    func attrType(name string) contentType {
140        name = strings.ToLower(name)

```

```

141     if strings.HasPrefix(name, "data-") {
142         // Strip data- so that custom attribute heur
143         // widely applied.
144         // Treat data-action as URL below.
145         name = name[5:]
146     } else if colon := strings.IndexRune(name, ':'); col
147         if name[:colon] == "xmlns" {
148             return contentTypeURL
149         }
150         // Treat svg:href and xlink:href as href bel
151         name = name[colon+1:]
152     }
153     if t, ok := attrTypeMap[name]; ok {
154         return t
155     }
156     // Treat partial event handler names as script.
157     if strings.HasPrefix(name, "on") {
158         return contentTypeJS
159     }
160
161     // Heuristics to prevent "javascript:..." injection
162     // data attributes and custom attributes like g:tweet
163     // http://www.w3.org/TR/html5/elements.html#embeddin
164     // "Custom data attributes are intended to store cus
165     // private to the page or application, for which th
166     // more appropriate attributes or elements."
167     // Developers seem to store URL content in data URLs
168     // or end with "URI" or "URL".
169     if strings.Contains(name, "src") ||
170         strings.Contains(name, "uri") ||
171         strings.Contains(name, "url") {
172         return contentTypeURL
173     }
174     return contentTypePlain
175 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/content.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "fmt"
9     "reflect"
10 )
11
12 // Strings of content from a trusted source.
13 type (
14     // CSS encapsulates known safe content that matches
15     // 1. The CSS3 stylesheet production, such as `p {
16     // 2. The CSS3 rule production, such as `a[href=~"
17     // 3. CSS3 declaration productions, such as `color
18     // 4. The CSS3 value production, such as `rgba(0,
19     // See http://www.w3.org/TR/css3-syntax/#style
20     CSS string
21
22     // HTML encapsulates a known safe HTML document frag
23     // It should not be used for HTML from a third-party
24     // unclosed tags or comments. The outputs of a sound
25     // and a template escaped by this package are fine f
26     HTML string
27
28     // HTMLAttr encapsulates an HTML attribute from a tr
29     // for example, ` dir="ltr"`.
30     HTMLAttr string
31
32     // JS encapsulates a known safe EcmaScript5 Expressi
33     // `(x + y * z())`.
34     // Template authors are responsible for ensuring tha
35     // do not break the intended precedence and that the
36     // statement/expression ambiguity as when passing an
37     // "{ foo: bar() }\n['foo']()", which is both a vali
38     // valid Program with a very different meaning.
39     JS string
40
41     // JSStr encapsulates a sequence of characters meant
```

```

42         // between quotes in a JavaScript expression.
43         // The string must match a series of StringCharacter
44         //   StringCharacter :: SourceCharacter but not ``
45         //                       | EscapeSequence
46         // Note that LineContinuations are not allowed.
47         // JSStr("foo\\nbar") is fine, but JSStr("foo\\\nbar
48         JSStr string
49
50         // URL encapsulates a known safe URL as defined in R
51         // A URL like `javascript:checkThatFormNotEditedBefo
52         // from a trusted source should go in the page, but
53         // `javascript:` URLs are filtered out since they ar
54         // exploited injection vector.
55         URL string
56     )
57
58     type contentType uint8
59
60     const (
61         contentTypePlain contentType = iota
62         contentTypeCSS
63         contentTypeHTML
64         contentTypeHTMLAttr
65         contentTypeJS
66         contentTypeJSStr
67         contentTypeURL
68         // contentTypeUnsafe is used in attr.go for values t
69         // embedded content and network messages are formed,
70         // or interpreted; or which credentials network mess
71         contentTypeUnsafe
72     )
73
74     // indirect returns the value, after dereferencing as many t
75     // as necessary to reach the base type (or nil).
76     func indirect(a interface{}) interface{} {
77         if t := reflect.TypeOf(a); t.Kind() != reflect.Ptr {
78             // Avoid creating a reflect.Value if it's no
79             return a
80         }
81         v := reflect.ValueOf(a)
82         for v.Kind() == reflect.Ptr && !v.IsNil() {
83             v = v.Elem()
84         }
85         return v.Interface()
86     }
87
88     var (
89         errorType          = reflect.TypeOf((*error)(nil)).Elem
90         fmtStringerType    = reflect.TypeOf((*fmt.Stringer)(nil)
91     )

```

```

92
93 // indirectToStringerOrError returns the value, after derefe
94 // as necessary to reach the base type (or nil) or an implem
95 // or error,
96 func indirectToStringerOrError(a interface{}) interface{} {
97     v := reflect.ValueOf(a)
98     for !v.Type().Implements(fmtStringerType) && !v.Type
99         v = v.Elem()
100     }
101     return v.Interface()
102 }
103
104 // stringify converts its arguments to a string and the type
105 // All pointers are dereferenced, as in the text/template pa
106 func stringify(args ...interface{}) (string, contentType) {
107     if len(args) == 1 {
108         switch s := indirect(args[0]).(type) {
109             case string:
110                 return s, contentTypePlain
111             case CSS:
112                 return string(s), contentTypeCSS
113             case HTML:
114                 return string(s), contentTypeHTML
115             case HTMLAttr:
116                 return string(s), contentTypeHTMLAtt
117             case JS:
118                 return string(s), contentTypeJS
119             case JSStr:
120                 return string(s), contentTypeJSStr
121             case URL:
122                 return string(s), contentTypeURL
123         }
124     }
125     for i, arg := range args {
126         args[i] = indirectToStringerOrError(arg)
127     }
128     return fmt.Sprint(args...), contentTypePlain
129 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/context.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "fmt"
9 )
10
11 // context describes the state an HTML parser must be in whe
12 // portion of HTML produced by evaluating a particular templ
13 //
14 // The zero value of type context is the start context for a
15 // produces an HTML fragment as defined at
16 // http://www.w3.org/TR/html5/the-end.html#parsing-html-frag
17 // where the context element is null.
18 type context struct {
19     state     state
20     delim    delim
21     urlPart  urlPart
22     jsCtx    jsCtx
23     attr     attr
24     element  element
25     err      *Error
26 }
27
28 func (c context) String() string {
29     return fmt.Sprintf("{%v %v %v %v %v %v %v}", c.state
30 }
31
32 // eq returns whether two contexts are equal.
33 func (c context) eq(d context) bool {
34     return c.state == d.state &&
35         c.delim == d.delim &&
36         c.urlPart == d.urlPart &&
37         c.jsCtx == d.jsCtx &&
38         c.attr == d.attr &&
39         c.element == d.element &&
40         c.err == d.err
41 }
```

```

42
43 // mangle produces an identifier that includes a suffix that
44 // from template names mangled with different contexts.
45 func (c context) mangle(templateName string) string {
46     // The mangled name for the default context is the i
47     if c.state == stateText {
48         return templateName
49     }
50     s := templateName + "$htmltemplate_" + c.state.Strin
51     if c.delim != 0 {
52         s += "_" + c.delim.String()
53     }
54     if c.urlPart != 0 {
55         s += "_" + c.urlPart.String()
56     }
57     if c.jsCtx != 0 {
58         s += "_" + c.jsCtx.String()
59     }
60     if c.attr != 0 {
61         s += "_" + c.attr.String()
62     }
63     if c.element != 0 {
64         s += "_" + c.element.String()
65     }
66     return s
67 }
68
69 // state describes a high-level HTML parser state.
70 //
71 // It bounds the top of the element stack, and by extension
72 // mode, but also contains state that does not correspond to
73 // HTML5 parsing algorithm because a single token production
74 // grammar may contain embedded actions in a template. For i
75 // HTML attribute produced by
76 //     <div title="Hello {{.World}}">
77 // is a single token in HTML's grammar but in a template spa
78 type state uint8
79
80 const (
81     // stateText is parsed character data. An HTML parse
82     // this state when its parse position is outside an
83     // directive, comment, and special element body.
84     stateText state = iota
85     // stateTag occurs before an HTML attribute or the e
86     stateTag
87     // stateAttrName occurs inside an attribute name.
88     // It occurs between the ^'s in `^name^ = value`.
89     stateAttrName
90     // stateAfterName occurs after an attr name has ende
91     // equals sign. It occurs between the ^'s in `name^

```

```

92     stateAfterName
93     // stateBeforeValue occurs after the equals sign but
94     // It occurs between the ^'s in `name =^ ^value`.
95     stateBeforeValue
96     // stateHTMLCmt occurs inside an <!-- HTML comment -
97     stateHTMLCmt
98     // stateRCDATA occurs inside an RCDATA element (<tex
99     // as described at http://dev.w3.org/html5/spec/synt
100    stateRCDATA
101    // stateAttr occurs inside an HTML attribute whose c
102    stateAttr
103    // stateURL occurs inside an HTML attribute whose co
104    stateURL
105    // stateJS occurs inside an event handler or script
106    stateJS
107    // stateJSDqStr occurs inside a JavaScript double qu
108    stateJSDqStr
109    // stateJSSqStr occurs inside a JavaScript single qu
110    stateJSSqStr
111    // stateJSRegexp occurs inside a JavaScript regexp l
112    stateJSRegexp
113    // stateJSBlockCmt occurs inside a JavaScript /* blo
114    stateJSBlockCmt
115    // stateJSLineCmt occurs inside a JavaScript // line
116    stateJSLineCmt
117    // stateCSS occurs inside a <style> element or style
118    stateCSS
119    // stateCSSDqStr occurs inside a CSS double quoted s
120    stateCSSDqStr
121    // stateCSSSqStr occurs inside a CSS single quoted s
122    stateCSSSqStr
123    // stateCSSDqURL occurs inside a CSS double quoted u
124    stateCSSDqURL
125    // stateCSSSqURL occurs inside a CSS single quoted u
126    stateCSSSqURL
127    // stateCSSURL occurs inside a CSS unquoted url(...)
128    stateCSSURL
129    // stateCSSBlockCmt occurs inside a CSS /* block com
130    stateCSSBlockCmt
131    // stateCSSLineCmt occurs inside a CSS // line comme
132    stateCSSLineCmt
133    // stateError is an infectious error state outside a
134    // HTML/CSS/JS construct.
135    stateError
136 )
137
138 var stateNames = [...]string{
139     stateText:     "stateText",
140     stateTag:      "stateTag",

```

```

141     stateAttrName:    "stateAttrName",
142     stateAfterName:   "stateAfterName",
143     stateBeforeValue: "stateBeforeValue",
144     stateHTMLCmt:    "stateHTMLCmt",
145     stateRCDATA:     "stateRCDATA",
146     stateAttr:       "stateAttr",
147     stateURL:        "stateURL",
148     stateJS:         "stateJS",
149     stateJSDqStr:    "stateJSDqStr",
150     stateJSSqStr:    "stateJSSqStr",
151     stateJSRegex:    "stateJSRegex",
152     stateJSBlockCmt: "stateJSBlockCmt",
153     stateJSLineCmt:  "stateJSLineCmt",
154     stateCSS:        "stateCSS",
155     stateCSSDqStr:   "stateCSSDqStr",
156     stateCSSSqStr:   "stateCSSSqStr",
157     stateCSSDqURL:   "stateCSSDqURL",
158     stateCSSSqURL:   "stateCSSSqURL",
159     stateCSSURL:     "stateCSSURL",
160     stateCSSBlockCmt: "stateCSSBlockCmt",
161     stateCSSLineCmt: "stateCSSLineCmt",
162     stateError:      "stateError",
163 }
164
165 func (s state) String() string {
166     if int(s) < len(stateNames) {
167         return stateNames[s]
168     }
169     return fmt.Sprintf("illegal state %d", int(s))
170 }
171
172 // isComment is true for any state that contains content mea
173 // authors & maintainers, not for end-users or machines.
174 func isComment(s state) bool {
175     switch s {
176     case stateHTMLCmt, stateJSBlockCmt, stateJSLineCmt,
177         return true
178     }
179     return false
180 }
181
182 // isInTag return whether s occurs solely inside an HTML tag
183 func isInTag(s state) bool {
184     switch s {
185     case stateTag, stateAttrName, stateAfterName, stateB
186         return true
187     }
188     return false
189 }

```

```

190
191 // delim is the delimiter that will end the current HTML att
192 type delim uint8
193
194 const (
195     // delimNone occurs outside any attribute.
196     delimNone delim = iota
197     // delimDoubleQuote occurs when a double quote (") c
198     delimDoubleQuote
199     // delimSingleQuote occurs when a single quote (') c
200     delimSingleQuote
201     // delimSpaceOrTagEnd occurs when a space or right a
202     // closes the attribute.
203     delimSpaceOrTagEnd
204 )
205
206 var delimNames = [...]string{
207     delimNone:      "delimNone",
208     delimDoubleQuote: "delimDoubleQuote",
209     delimSingleQuote: "delimSingleQuote",
210     delimSpaceOrTagEnd: "delimSpaceOrTagEnd",
211 }
212
213 func (d delim) String() string {
214     if int(d) < len(delimNames) {
215         return delimNames[d]
216     }
217     return fmt.Sprintf("illegal delim %d", int(d))
218 }
219
220 // urlPart identifies a part in an RFC 3986 hierarchical URL
221 // encoding strategies.
222 type urlPart uint8
223
224 const (
225     // urlPartNone occurs when not in a URL, or possibly
226     // ^ in "^http://auth/path?k=v#frag".
227     urlPartNone urlPart = iota
228     // urlPartPreQuery occurs in the scheme, authority,
229     // ^s in "h^http://auth/path^?k=v#frag".
230     urlPartPreQuery
231     // urlPartQueryOrFrag occurs in the query portion be
232     // "http://auth/path?^k=v#frag^".
233     urlPartQueryOrFrag
234     // urlPartUnknown occurs due to joining of contexts
235     // after the query separator.
236     urlPartUnknown
237 )
238
239 var urlPartNames = [...]string{

```

```

240         urlPartNone:         "urlPartNone",
241         urlPartPreQuery:     "urlPartPreQuery",
242         urlPartQueryOrFrag:  "urlPartQueryOrFrag",
243         urlPartUnknown:     "urlPartUnknown",
244     }
245
246     func (u urlPart) String() string {
247         if int(u) < len(urlPartNames) {
248             return urlPartNames[u]
249         }
250         return fmt.Sprintf("illegal urlPart %d", int(u))
251     }
252
253     // jsCtx determines whether a '/' starts a regular expressio
254     // division operator.
255     type jsCtx uint8
256
257     const (
258         // jsCtxRegexp occurs where a '/' would start a rege
259         jsCtxRegexp jsCtx = iota
260         // jsCtxDivOp occurs where a '/' would start a divis
261         jsCtxDivOp
262         // jsCtxUnknown occurs where a '/' is ambiguous due
263         jsCtxUnknown
264     )
265
266     func (c jsCtx) String() string {
267         switch c {
268             case jsCtxRegexp:
269                 return "jsCtxRegexp"
270             case jsCtxDivOp:
271                 return "jsCtxDivOp"
272             case jsCtxUnknown:
273                 return "jsCtxUnknown"
274         }
275         return fmt.Sprintf("illegal jsCtx %d", int(c))
276     }
277
278     // element identifies the HTML element when inside a start t
279     // Certain HTML element (for example <script> and <style>) h
280     // treated differently from stateText so the element type is
281     // transition into the correct context at the end of a tag a
282     // end delimiter for the body.
283     type element uint8
284
285     const (
286         // elementNone occurs outside a special tag or speci
287         elementNone element = iota
288         // elementScript corresponds to the raw text <script

```

```

289     elementScript
290     // elementStyle corresponds to the raw text <style>
291     elementStyle
292     // elementTextarea corresponds to the RCDATA <textar
293     elementTextarea
294     // elementTitle corresponds to the RCDATA <title> el
295     elementTitle
296 )
297
298 var elementNames = [...]string{
299     elementNone:    "elementNone",
300     elementScript:  "elementScript",
301     elementStyle:   "elementStyle",
302     elementTextarea: "elementTextarea",
303     elementTitle:   "elementTitle",
304 }
305
306 func (e element) String() string {
307     if int(e) < len(elementNames) {
308         return elementNames[e]
309     }
310     return fmt.Sprintf("illegal element %d", int(e))
311 }
312
313 // attr identifies the most recent HTML attribute when insid
314 type attr uint8
315
316 const (
317     // attrNone corresponds to a normal attribute or no
318     attrNone attr = iota
319     // attrScript corresponds to an event handler attrib
320     attrScript
321     // attrStyle corresponds to the style attribute whos
322     attrStyle
323     // attrURL corresponds to an attribute whose value i
324     attrURL
325 )
326
327 var attrNames = [...]string{
328     attrNone:    "attrNone",
329     attrScript:  "attrScript",
330     attrStyle:   "attrStyle",
331     attrURL:     "attrURL",
332 }
333
334 func (a attr) String() string {
335     if int(a) < len(attrNames) {
336         return attrNames[a]
337     }

```

```
338         return fmt.Sprintf("illegal attr %d", int(a))
339     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/css.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "bytes"
9     "fmt"
10    "unicode"
11    "unicode/utf8"
12 )
13
14 // endsWithCSSKeyword returns whether b ends with an ident t
15 // case-insensitively matches the lower-case kw.
16 func endsWithCSSKeyword(b []byte, kw string) bool {
17     i := len(b) - len(kw)
18     if i < 0 {
19         // Too short.
20         return false
21     }
22     if i != 0 {
23         r, _ := utf8.DecodeLastRune(b[:i])
24         if isCSSNmchar(r) {
25             // Too long.
26             return false
27         }
28     }
29     // Many CSS keywords, such as "!important" can have
30     // but the URI production does not allow that accord
31     // http://www.w3.org/TR/css3-syntax/#TOK-URI
32     // This does not attempt to recognize encoded keywor
33     // given "\75\72\6c" and "url" this return false.
34     return string(bytes.ToLower(b[i:])) == kw
35 }
36
37 // isCSSNmchar returns whether rune is allowed anywhere in a
38 func isCSSNmchar(r rune) bool {
39     // Based on the CSS3 nmchar production but ignores r
40     // sequences.
41     // http://www.w3.org/TR/css3-syntax/#SUBTOK-nmchar
```

```

42     return 'a' <= r && r <= 'z' ||
43           'A' <= r && r <= 'Z' ||
44           '0' <= r && r <= '9' ||
45           r == '-' ||
46           r == '_' ||
47           // Non-ASCII cases below.
48           0x80 <= r && r <= 0xd7ff ||
49           0xe000 <= r && r <= 0xffffd ||
50           0x10000 <= r && r <= 0x10ffff
51 }
52
53 // decodeCSS decodes CSS3 escapes given a sequence of string
54 // If there is no change, it returns the input, otherwise it
55 // backed by a new array.
56 // http://www.w3.org/TR/css3-syntax/#SUBTOK-stringchar defin
57 func decodeCSS(s []byte) []byte {
58     i := bytes.IndexByte(s, '\\')
59     if i == -1 {
60         return s
61     }
62     // The UTF-8 sequence for a codepoint is never longer
63     // number hex digits need to represent that codepoint
64     // upper bound on the output length.
65     b := make([]byte, 0, len(s))
66     for len(s) != 0 {
67         i := bytes.IndexByte(s, '\\')
68         if i == -1 {
69             i = len(s)
70         }
71         b, s = append(b, s[:i]...), s[i:]
72         if len(s) < 2 {
73             break
74         }
75         // http://www.w3.org/TR/css3-syntax/#SUBTOK-
76         // escape ::= unicode | '\\' [#x20-#x7E#x80-#
77         if isHex(s[1]) {
78             // http://www.w3.org/TR/css3-syntax/
79             // unicode ::= '\\' [0-9a-fA-F]{1,6
80             j := 2
81             for j < len(s) && j < 7 && isHex(s[j]
82                 j++
83             }
84             r := hexDecode(s[1:j])
85             if r > unicode.MaxRune {
86                 r, j = r/16, j-1
87             }
88             n := utf8.EncodeRune(b[len(b):cap(b)]
89             // The optional space at the end all
90             // sequence to be followed by a lite
91             // string(decodeCSS([]byte(`A B`)))

```

```

92         b, s = b[:len(b)+n], skipCSSSpace(s[
93     } else {
94         // `\\` decodes to `` and `\"` to `
95         _, n := utf8.DecodeRune(s[1:])
96         b, s = append(b, s[1:1+n]...), s[1+n
97     }
98 }
99     return b
100 }
101
102 // isHex returns whether the given character is a hex digit.
103 func isHex(c byte) bool {
104     return '0' <= c && c <= '9' || 'a' <= c && c <= 'f'
105 }
106
107 // hexDecode decodes a short hex digit sequence: "10" -> 16.
108 func hexDecode(s []byte) rune {
109     n := '\x00'
110     for _, c := range s {
111         n <= 4
112         switch {
113         case '0' <= c && c <= '9':
114             n |= rune(c - '0')
115         case 'a' <= c && c <= 'f':
116             n |= rune(c-'a') + 10
117         case 'A' <= c && c <= 'F':
118             n |= rune(c-'A') + 10
119         default:
120             panic(fmt.Sprintf("Bad hex digit in
121         })
122     }
123     return n
124 }
125
126 // skipCSSSpace returns a suffix of c, skipping over a single
127 func skipCSSSpace(c []byte) []byte {
128     if len(c) == 0 {
129         return c
130     }
131     // wc ::= #x9 | #xA | #xC | #xD | #x20
132     switch c[0] {
133     case '\t', '\n', '\f', ' ':
134         return c[1:]
135     case '\r':
136         // This differs from CSS3's wc production because
137         // probable spec error whereby wc contains a
138         // sequences in nl (newline) but not CRLF.
139         if len(c) >= 2 && c[1] == '\n' {
140             return c[2:]

```

```

141         }
142         return c[1:]
143     }
144     return c
145 }
146
147 // isCSSSpace returns whether b is a CSS space char as defin
148 func isCSSSpace(b byte) bool {
149     switch b {
150     case '\t', '\n', '\f', '\r', ' ':
151         return true
152     }
153     return false
154 }
155
156 // cssEscaper escapes HTML and CSS special characters using
157 func cssEscaper(args ...interface{}) string {
158     s, _ := stringify(args...)
159     var b bytes.Buffer
160     written := 0
161     for i, r := range s {
162         var repl string
163         switch r {
164         case 0:
165             repl = `\0`
166         case '\t':
167             repl = `\9`
168         case '\n':
169             repl = `\a`
170         case '\f':
171             repl = `\c`
172         case '\r':
173             repl = `\d`
174         // Encode HTML specials as hex so the output
175         // in HTML attributes without further encodi
176         case '"':
177             repl = `\22`
178         case '&':
179             repl = `\26`
180         case '\\':
181             repl = `\27`
182         case '(':
183             repl = `\28`
184         case ')':
185             repl = `\29`
186         case '+':
187             repl = `\2b`
188         case '/':
189             repl = `\2f`

```

```

190         case ':':
191             repl = `\\3a`
192         case ';':
193             repl = `\\3b`
194         case '<':
195             repl = `\\3c`
196         case '>':
197             repl = `\\3e`
198         case '\\':
199             repl = `\\`
200         case '{':
201             repl = `\\7b`
202         case '}':
203             repl = `\\7d`
204         default:
205             continue
206     }
207     b.WriteString(s[written:i])
208     b.WriteString(repl)
209     written = i + utf8.RuneLen(r)
210     if repl != `\\` && (written == len(s) || isH
211         b.WriteByte(' ')
212     }
213 }
214 if written == 0 {
215     return s
216 }
217 b.WriteString(s[written:])
218 return b.String()
219 }
220
221 var expressionBytes = []byte("expression")
222 var mozBindingBytes = []byte("mozbinding")
223
224 // cssValueFilter allows innocuous CSS values in the output
225 // quantities (10px or 25%), ID or class literals (#foo, .ba
226 // (inherit, blue), and colors (#888).
227 // It filters out unsafe values, such as those that affect t
228 // and anything that might execute scripts.
229 func cssValueFilter(args ...interface{}) string {
230     s, t := stringify(args...)
231     if t == contentTypeCSS {
232         return s
233     }
234     b, id := decodeCSS([]byte(s)), make([]byte, 0, 64)
235
236     // CSS3 error handling is specified as honoring stri
237     // http://www.w3.org/TR/css3-syntax/#error-handling
238     // Malformed declarations. User agents must hand
239     // tokens encountered while parsing a declaratio

```

```

240 // the end of the declaration, while observing t
241 // matching pairs of (), [], {}, "", and ', and
242 // escapes. For example, a malformed declaration
243 // property, colon (:) or value.
244 // So we need to make sure that values do not have m
245 // or quote characters to prevent the browser from r
246 // inside a string that might embed JavaScript sourc
247 for i, c := range b {
248     switch c {
249     case 0, '"', '\\', '(', ')', '/', ';', '@',
250         return filterFailsafe
251     case '-':
252         // Disallow <!-- or -->.
253         // -- should not appear in valid ide
254         if i != 0 && b[i-1] == '-' {
255             return filterFailsafe
256         }
257     default:
258         if c < 0x80 && isCSSNmchar(rune(c))
259             id = append(id, c)
260         }
261     }
262 }
263 id = bytes.ToLower(id)
264 if bytes.Index(id, expressionBytes) != -1 || bytes.I
265     return filterFailsafe
266 }
267 return string(b)
268 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/doc.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6 Package template (html/template) implements data-driven temp
7 generating HTML output safe against code injection. It provi
8 same interface as package text/template and should be used i
9 text/template whenever the output is HTML.
10
11 The documentation here focuses on the security features of t
12 For information about how to program the templates themselve
13 documentation for text/template.
14
15 Introduction
16
17 This package wraps package text/template so you can share it
18 to parse and execute HTML templates safely.
19
20     tmpl, err := template.New("name").Parse(...)
21     // Error checking elided
22     err = tmpl.Execute(out, data)
23
24 If successful, tmpl will now be injection-safe. Otherwise, e
25 defined in the docs for ErrorCode.
26
27 HTML templates treat data values as plain text which should
28 can be safely embedded in an HTML document. The escaping is
29 actions can appear within JavaScript, CSS, and URI contexts.
30
31 The security model used by this package assumes that templat
32 trusted, while Execute's data parameter is not. More details
33 provided below.
34
35 Example
36
37     import "text/template"
38     ...
39     t, err := template.New("foo").Parse(`{{define "T"}}Hello,
40     err = t.ExecuteTemplate(out, "T", "<script>alert('you have
41
```



```

92     <a href="?q={{.}}">                                O&#39;Reilly%3a%20How%20a
93     <a onx='f("{{.}}")'>                                O\x27Reilly: How are \x3c
94     <a onx='f({{.}})'>                                  "O\x27Reilly: How are \x3
95     <a onx='pattern = /{{.}}/;'>                        O\x27Reilly: How are \x3c
96
97 If used in an unsafe context, then the value might be filter
98
99 Context                                                {{.}} After
100 <a href="{{.}}">                                        #ZgotmplZ
101
102 since "O'Reilly:" is not an allowed protocol like "http:".
103
104
105 If {{.}} is the innocuous word, `left`, then it can appear r
106
107 Context                                                {{.}} After
108 {{.}}                                                  left
109 <a title='{{.}}'>                                       left
110 <a href='{{.}}'>                                         left
111 <a href='/{{.}}'>                                       left
112 <a href='?dir={{.}}'>                                       left
113 <a style="border-{{.}}: 4px">                                       left
114 <a style="align: {{.}}">                                       left
115 <a style="background: '{{.}}'>                                       left
116 <a style="background: url('{{.}}')> left
117 <style>p.{{.}} {color:red}</style> left
118
119 Non-string values can be used in JavaScript contexts.
120 If {{.}} is
121
122     []struct{A,B string}{ "foo", "bar" }
123
124 in the escaped template
125
126     <script>var pair = {{.}};</script>
127
128 then the template output is
129
130     <script>var pair = {"A": "foo", "B": "bar"};</script>
131
132 See package json to understand how non-string content is mar
133 embedding in JavaScript contexts.
134
135
136 Typed Strings
137
138 By default, this package assumes that all pipelines produce
139 It adds escaping pipeline stages necessary to correctly and
140 plain text string in the appropriate context.

```

141
142 When a data value is not plain text, you can make sure it is
143 by marking it with its type.
144
145 Types HTML, JS, URL, and others from content.go can carry sa
146 exempted from escaping.
147
148 The template
149 Hello, {{.}}!
151
152 can be invoked with
153 tmpl.Execute(out, HTML(`**World**`))
155
156 to produce
157 Hello, **World**!
159
160 instead of the
161 Hello, World!
163
164 that would have been produced if {{.}} was a regular string.
165
166
167 Security Model
168
169 <http://js-quasis-libraries-and-repl.googlecode.com/svn/trunk>
170
171 This package assumes that template authors are trusted, that
172 parameter is not, and seeks to preserve the properties below
173 of untrusted data:
174
175 Structure Preservation Property:
176 "... when a template author writes an HTML tag in a safe tem
177 the browser will interpret the corresponding portion of the
178 regardless of the values of untrusted data, and similarly fo
179 such as attribute boundaries and JS and CSS string boundarie
180
181 Code Effect Property:
182 "... only code specified by the template author should run a
183 injecting the template output into a page and all code speci
184 template author should run as a result of the same."
185
186 Least Surprise Property:
187 "A developer (or code reviewer) familiar with HTML, CSS, and
188 knows that contextual autoescaping happens should be able to
189 and correctly infer what sanitization happens."

```
190 */  
191 package template
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/error.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "fmt"
9 )
10
11 // Error describes a problem encountered during template Esc
12 type Error struct {
13     // ErrorCode describes the kind of error.
14     ErrorCode ErrorCode
15     // Name is the name of the template in which the err
16     Name string
17     // Line is the line number of the error in the templ
18     Line int
19     // Description is a human-readable description of th
20     Description string
21 }
22
23 // ErrorCode is a code for a kind of error.
24 type ErrorCode int
25
26 // We define codes for each error that manifests while escap
27 // escaped templates may also fail at runtime.
28 //
29 // Output: "ZgotmplZ"
30 // Example:
31 //   
32 //   where {{.X}} evaluates to `javascript:...`
33 // Discussion:
34 //   "ZgotmplZ" is a special value that indicates that unsaf
35 //   CSS or URL context at runtime. The output of the exampl
36 //   
37 //   If the data comes from a trusted source, use content ty
38 //   from filtering: URL(`javascript:...`).
39 const (
40     // OK indicates the lack of an error.
41     OK ErrorCode = iota
```

```

42
43 // ErrAmbigContext: "... appears in an ambiguous URL
44 // Example:
45 //   <a href="
46 //     {{if .C}}
47 //       /path/
48 //     {{else}}
49 //       /search?q=
50 //     {{end}}
51 //     {{.X}}
52 //   ">
53 // Discussion:
54 //   {{.X}} is in an ambiguous URL context since, de
55 //   it may be either a URL suffix or a query paramet
56 //   Moving {{.X}} into the condition removes the ar
57 //   <a href="{{if .C}}/path/{{.X}}{{else}}/search?q
58 ErrAmbigContext
59
60 // ErrBadHTML: "expected space, attr name, or end of
61 // "... in unquoted attr", "... in attribute name"
62 // Example:
63 //   <a href = /search?q=foo>
64 //   <href=foo>
65 //   <form na<e=...>
66 //   <option selected<
67 // Discussion:
68 //   This is often due to a typo in an HTML element,
69 //   are banned in tag names, attribute names, and u
70 //   values because they can tickle parser ambiguiti
71 //   Quoting all attributes is the best policy.
72 ErrBadHTML
73
74 // ErrBranchEnd: "{{if}} branches end in different c
75 // Example:
76 //   {{if .C}}<a href="{{end}} {{.X}}
77 // Discussion:
78 //   Package html/template statically examines each
79 //   {{if}}, {{range}}, or {{with}} to escape any fo
80 //   The example is ambiguous since {{.X}} might be
81 //   or a URL prefix in an HTML attribute. The conte
82 //   used to figure out how to escape it, but that c
83 //   the run-time value of {{.C}} which is not stati
84 //
85 //   The problem is usually something like missing q
86 //   brackets, or can be avoided by refactoring to p
87 //   into different branches of an if, range or with
88 //   is in a {{range}} over a collection that should
89 //   adding a dummy {{else}} can help.
90 ErrBranchEnd
91

```

```

92 // ErrEndContext: "... ends in a non-text context: ."
93 // Examples:
94 // <div
95 // <div title="no close quote>
96 // <script>f()
97 // Discussion:
98 // Executed templates should produce a DocumentFra
99 // Templates that end without closing tags will tr
100 // Templates that should not be used in an HTML co
101 // produce incomplete Fragments should not be exec
102 //
103 // {{define "main"}} <script>{{template "helper"}}
104 // {{define "helper"}} document.write(' <div title
105 //
106 // "helper" does not produce a valid document frag
107 // not be Executed directly.
108 ErrEndContext
109
110 // ErrNoSuchTemplate: "no such template ..."
111 // Examples:
112 // {{define "main"}}<div {{template "attrs"}}>{{en
113 // {{define "attrs"}}href="{{.URL}}"{{end}}
114 // Discussion:
115 // Package html/template looks through template ca
116 // context.
117 // Here the {{.URL}} in "attrs" must be treated as
118 // from "main", but you will get this error if "at
119 // when "main" is parsed.
120 ErrNoSuchTemplate
121
122 // ErrOutputContext: "cannot compute output context
123 // Examples:
124 // {{define "t"}}{{if .T}}{{template "t" .T}}{{end
125 // Discussion:
126 // A recursive template does not end in the same c
127 // starts, and a reliable output context cannot be
128 // Look for typos in the named template.
129 // If the template should not be called in the nar
130 // look for calls to that template in unexpected c
131 // Maybe refactor recursive templates to not be re
132 ErrOutputContext
133
134 // ErrPartialCharset: "unfinished JS regexp charset
135 // Example:
136 // <script>var pattern = /foo[{{.Chars}}]/</scri
137 // Discussion:
138 // Package html/template does not support interpol
139 // expression literal character sets.
140 ErrPartialCharset

```

```

141
142 // ErrPartialEscape: "unfinished escape sequence in
143 // Example:
144 // <script>alert("\{{.X}}")</script>
145 // Discussion:
146 // Package html/template does not support actions
147 // backslash.
148 // This is usually an error and there are better s
149 // example
150 // <script>alert("{{.X}}")</script>
151 // should work, and if {{.X}} is a partial escape
152 // "xA0", mark the whole sequence as safe content:
153 ErrPartialEscape
154
155 // ErrRangeLoopReentry: "on range loop re-entry: ...
156 // Example:
157 // <script>var x = [{{range .}}'{{.}},{{end}}]</sc
158 // Discussion:
159 // If an iteration through a range would cause it
160 // different context than an earlier pass, there i
161 // In the example, there is missing a quote, so it
162 // whether {{.}} is meant to be inside a JS string
163 // context. The second iteration would produce so
164 //
165 // <script>var x = ['firstValue,'secondValue]</s
166 ErrRangeLoopReentry
167
168 // ErrSlashAmbig: '/' could start a division or rege
169 // Example:
170 // <script>
171 //     {{if .C}}var x = 1{{end}}
172 //     /-{{.N}}/i.test(x) ? doThis : doThat();
173 // </script>
174 // Discussion:
175 // The example above could produce `var x = 1/-2/i
176 // in which the first '/' is a mathematical divisi
177 // could produce `/-2/i.test(s)` in which the firs
178 // regexp literal.
179 // Look for missing semicolons inside branches, an
180 // parentheses to make it clear which interpretati
181 ErrSlashAmbig
182 )
183
184 func (e *Error) Error() string {
185     if e.Line != 0 {
186         return fmt.Sprintf("html/template:%s:%d: %s"
187     } else if e.Name != "" {
188         return fmt.Sprintf("html/template:%s: %s", e
189     }

```

```
190         return "html/template: " + e.Description
191     }
192
193     // errorf creates an error given a format string f and args.
194     // The template Name still needs to be supplied.
195     func errorf(k ErrorCode, line int, f string, args ...interfa
196         return &Error{k, "", line, fmt.Sprintf(f, args...)}
197     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/escape.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "bytes"
9     "fmt"
10    "html"
11    "io"
12    "text/template"
13    "text/template/parse"
14 )
15
16 // escapeTemplates rewrites the named templates, which must
17 // associated with t, to guarantee that the output of any of
18 // templates is properly escaped. Names should include the
19 // all templates that might be Executed but need not include
20 // templates. If no error is returned, then the named templ
21 // been modified. Otherwise the named templates have been r
22 // unusable.
23 func escapeTemplates(tmpl *Template, names ...string) error
24     e := newEscaper(tmpl)
25     for _, name := range names {
26         c, _ := e.escapeTree(context{}, name, 0)
27         var err error
28         if c.err != nil {
29             err, c.err.Name = c.err, name
30         } else if c.state != stateText {
31             err = &Error{ErrEndContext, name, 0,
32         }
33         if err != nil {
34             // Prevent execution of unsafe templ
35             for _, name := range names {
36                 if t := tmpl.set[name]; t !=
37                     t.text.Tree = nil
38             }
39         }
40         return err
41     }
```

```

42         tmpl.escaped = true
43     }
44     e.commit()
45     return nil
46 }
47
48 // funcMap maps command names to functions that render their
49 var funcMap = template.FuncMap{
50     "html_template_attrscaper":    attrEscaper,
51     "html_template_commentescaper": commentEscaper,
52     "html_template_cssescaper":    cssEscaper,
53     "html_template_cssvaluefilter": cssValueFilter,
54     "html_template_htmlnamefilter": htmlNameFilter,
55     "html_template_htmllescaper":  htmlEscaper,
56     "html_template_jsregexpescaper": jsRegexEscaper,
57     "html_template_jsstrescaper":  jsStrEscaper,
58     "html_template_jsvalescaper":  jsValEscaper,
59     "html_template_nospaceescaper": htmlNospaceEscaper,
60     "html_template_rcdataescaper":  rcdataEscaper,
61     "html_template_urlescaper":     urlEscaper,
62     "html_template_urlfilter":      urlFilter,
63     "html_template_urlnormalizer":  urlNormalizer,
64 }
65
66 // equivEscapers matches contextual escapers to equivalent t
67 var equivEscapers = map[string]string{
68     "html_template_attrscaper":    "html",
69     "html_template_htmllescaper":  "html",
70     "html_template_nospaceescaper": "html",
71     "html_template_rcdataescaper":  "html",
72     "html_template_urlescaper":     "urlquery",
73     "html_template_urlnormalizer":  "urlquery",
74 }
75
76 // escaper collects type inferences about templates and chan
77 // templates injection safe.
78 type escaper struct {
79     tmpl *Template
80     // output[templateName] is the output context for a
81     // has been mangled to include its input context.
82     output map[string]context
83     // derived[c.mangle(name)] maps to a template derive
84     // named name templateName for the start context c.
85     derived map[string]*template.Template
86     // called[templateName] is a set of called mangled t
87     // called map[string]bool
88     // xxxNodeEdits are the accumulated edits to apply d
89     // Such edits are not applied immediately in case a
90     // executes a given template in different escaping c
91     actionNodeEdits map[*parse.ActionNode][]string

```

```

92         templateNodeEdits map[*parse.TemplateNode]string
93         textNodeEdits     map[*parse.TextNode][]byte
94     }
95
96     // newEscaper creates a blank escaper for the given set.
97     func newEscaper(t *Template) *escaper {
98         return &escaper{
99             t,
100            map[string]context{},
101            map[string]*template.Template{},
102            map[string]bool{},
103            map[*parse.ActionNode][]string{},
104            map[*parse.TemplateNode]string{},
105            map[*parse.TextNode][]byte{},
106        }
107     }
108
109     // filterFailsafe is an innocuous word that is emitted in pl
110     // by sanitizer functions. It is not a keyword in any progra
111     // contains no special characters, is not empty, and when it
112     // it is distinct enough that a developer can find the sourc
113     // via a search engine.
114     const filterFailsafe = "ZgotmplZ"
115
116     // escape escapes a template node.
117     func (e *escaper) escape(c context, n parse.Node) context {
118         switch n := n.(type) {
119             case *parse.ActionNode:
120                 return e.escapeAction(c, n)
121             case *parse.IfNode:
122                 return e.escapeBranch(c, &n.BranchNode, "if"
123             case *parse.ListNode:
124                 return e.escapeList(c, n)
125             case *parse.RangeNode:
126                 return e.escapeBranch(c, &n.BranchNode, "ran
127             case *parse.TemplateNode:
128                 return e.escapeTemplate(c, n)
129             case *parse.TextNode:
130                 return e.escapeText(c, n)
131             case *parse.WithNode:
132                 return e.escapeBranch(c, &n.BranchNode, "wit
133         }
134         panic("escaping " + n.String() + " is unimplemented"
135     }
136
137     // escapeAction escapes an action template node.
138     func (e *escaper) escapeAction(c context, n *parse.ActionNode) context {
139         if len(n.Pipe.Decl) != 0 {
140             // A local variable assignment, not an inter

```

```

141         return c
142     }
143     c = nudge(c)
144     s := make([]string, 0, 3)
145     switch c.state {
146     case stateError:
147         return c
148     case stateURL, stateCSSDqStr, stateCSSSqStr, stateCS
149         switch c.urlPart {
150         case urlPartNone:
151             s = append(s, "html_template_urlfilt
152             fallthrough
153         case urlPartPreQuery:
154             switch c.state {
155             case stateCSSDqStr, stateCSSSqStr:
156                 s = append(s, "html_template
157             default:
158                 s = append(s, "html_template
159             }
160         case urlPartQueryOrFrag:
161             s = append(s, "html_template_urlesca
162         case urlPartUnknown:
163             return context{
164                 state: stateError,
165                 err:   errorf(ErrAmbigContex
166             }
167         default:
168             panic(c.urlPart.String())
169     }
170     case stateJS:
171         s = append(s, "html_template_jsvalescaper")
172         // A slash after a value starts a div operat
173         c.jsCtx = jsCtxDivOp
174     case stateJSDqStr, stateJSSqStr:
175         s = append(s, "html_template_jsstrescaper")
176     case stateJSRegexp:
177         s = append(s, "html_template_jsregexpescaper
178     case stateCSS:
179         s = append(s, "html_template_cssvaluefilter"
180     case stateText:
181         s = append(s, "html_template_htmlscaper")
182     case stateRCDATA:
183         s = append(s, "html_template_rcdataescaper")
184     case stateAttr:
185         // Handled below in delim check.
186     case stateAttrName, stateTag:
187         c.state = stateAttrName
188         s = append(s, "html_template_htmlnamefilter"
189     default:

```

```

190         if isComment(c.state) {
191             s = append(s, "html_template_comment
192         } else {
193             panic("unexpected state " + c.state.
194         }
195     }
196     switch c.delim {
197     case delimNone:
198         // No extra-escaping needed for raw text con
199     case delimSpaceOrTagEnd:
200         s = append(s, "html_template_nospaceescaper"
201     default:
202         s = append(s, "html_template_attrescaper")
203     }
204     e.editActionNode(n, s)
205     return c
206 }
207
208 // ensurePipelineContains ensures that the pipeline has comm
209 // the identifiers in s in order.
210 // If the pipeline already has some of the sanitizers, do no
211 // For example, if p is (.X | html) and s is ["escapeJSVal",
212 // has one matching, "html", and one to insert, "escapeJSVal
213 // (.X | escapeJSVal | html).
214 func ensurePipelineContains(p *parse.PipeNode, s []string) {
215     if len(s) == 0 {
216         return
217     }
218     n := len(p.Cmds)
219     // Find the identifiers at the end of the command ch
220     idents := p.Cmds
221     for i := n - 1; i >= 0; i-- {
222         if cmd := p.Cmds[i]; len(cmd.Args) != 0 {
223             if id, ok := cmd.Args[0].(*parse.Ide
224                 if id.Ident == "noescape" {
225                 return
226             }
227             continue
228         }
229     }
230     idents = p.Cmds[i+1:]
231 }
232 dups := 0
233 for _, id := range idents {
234     if escFnsEq(s[dups], (id.Args[0].(*parse.Ide
235         dups++
236         if dups == len(s) {
237             return
238         }
239     }

```

```

240     }
241     newCmds := make([]*parse.CommandNode, n-len(identfs),
242     copy(newCmds, p.Cmds)
243     // Merge existing identifier commands with the sanit
244     for _, id := range identfs {
245         i := indexOfStr((id.Args[0].(*parse.Identifi
246         if i != -1 {
247             for _, name := range s[:i] {
248                 newCmds = appendCmd(newCmds,
249                 }
250                 s = s[i+1:]
251             }
252             newCmds = appendCmd(newCmds, id)
253         }
254         // Create any remaining sanitizers.
255         for _, name := range s {
256             newCmds = appendCmd(newCmds, newIdentCmd(nam
257         }
258         p.Cmds = newCmds
259     }
260
261     // redundantFuncs[a][b] implies that funcMap[b](funcMap[a](x
262     // for all x.
263     var redundantFuncs = map[string]map[string]bool{
264         "html_template_commentescaper": {
265             "html_template_attrescaper": true,
266             "html_template_nospaceescaper": true,
267             "html_template_htmllescaper": true,
268         },
269         "html_template_cssescaper": {
270             "html_template_attrescaper": true,
271         },
272         "html_template_jsregexpescaper": {
273             "html_template_attrescaper": true,
274         },
275         "html_template_jsstrescaper": {
276             "html_template_attrescaper": true,
277         },
278         "html_template_urlescaper": {
279             "html_template_urlnormalizer": true,
280         },
281     }
282
283     // appendCmd appends the given command to the end of the com
284     // unless it is redundant with the last command.
285     func appendCmd(cmds []*parse.CommandNode, cmd *parse.Command
286         if n := len(cmds); n != 0 {
287             last, ok := cmds[n-1].Args[0].(*parse.Identi
288             next, _ := cmd.Args[0].(*parse.IdentifierNod

```

```

289             if ok && redundantFuncs[last.Ident][next.Ide
290                 return cmds
291             }
292         }
293         return append(cmds, cmd)
294     }
295
296     // indexOfStr is the first i such that eq(s, strs[i]) or -1
297     func indexOfStr(s string, strs []string, eq func(a, b string)
298         for i, t := range strs {
299             if eq(s, t) {
300                 return i
301             }
302         }
303         return -1
304     }
305
306     // escFnsEq returns whether the two escaping functions are e
307     func escFnsEq(a, b string) bool {
308         if e := equivEscapers[a]; e != "" {
309             a = e
310         }
311         if e := equivEscapers[b]; e != "" {
312             b = e
313         }
314         return a == b
315     }
316
317     // newIdentCmd produces a command containing a single identi
318     func newIdentCmd(identifier string) *parse.CommandNode {
319         return &parse.CommandNode{
320             NodeType: parse.NodeCommand,
321             Args:      []parse.Node{parse.NewIdentifier(i
322         }
323     }
324
325     // nudge returns the context that would result from followin
326     // transitions from the input context.
327     // For example, parsing:
328     //     `<a href=`
329     // will end in context{stateBeforeValue, attrURL}, but parsi
330     //     `<a href=x`
331     // will end in context{stateURL, delimSpaceOrTagEnd, ...}.
332     // There are two transitions that happen when the 'x' is see
333     // (1) Transition from a before-value state to a start-of-va
334     //     consuming any character.
335     // (2) Consume 'x' and transition past the first value chara
336     // In this case, nudging produces the context after (1) happ
337     func nudge(c context) context {

```

```

338     switch c.state {
339     case stateTag:
340         // In `<foo {{.}}`, the action should emit a
341         c.state = stateAttrName
342     case stateBeforeValue:
343         // In `<foo bar={{.}}`, the action is an und
344         c.state, c.delim, c.attr = attrStartStates[c
345     case stateAfterName:
346         // In `<foo bar {{.}}`, the action is an att
347         c.state, c.attr = stateAttrName, attrNone
348     }
349     return c
350 }
351
352 // join joins the two contexts of a branch template node. Th
353 // error context if either of the input contexts are error c
354 // the input contexts differ.
355 func join(a, b context, line int, nodeName string) context {
356     if a.state == stateError {
357         return a
358     }
359     if b.state == stateError {
360         return b
361     }
362     if a.eq(b) {
363         return a
364     }
365
366     c := a
367     c.urlPart = b.urlPart
368     if c.eq(b) {
369         // The contexts differ only by urlPart.
370         c.urlPart = urlPartUnknown
371         return c
372     }
373
374     c = a
375     c.jsCtx = b.jsCtx
376     if c.eq(b) {
377         // The contexts differ only by jsCtx.
378         c.jsCtx = jsCtxUnknown
379         return c
380     }
381
382     // Allow a nudged context to join with an unnudged c
383     // This means that
384     // <p title={{if .C}}{{.}}{{end}}
385     // ends in an unquoted value state even though the e
386     // ends in stateBeforeValue.
387     if c, d := nudge(a), nudge(b); !(c.eq(a) && d.eq(b))

```

```

388         if e := join(c, d, line, nodeName); e.state
389             return e
390     }
391 }
392
393 return context{
394     state: stateError,
395     err:   errorf(ErrBranchEnd, line, "{{%s}} br
396 }
397 }
398
399 // escapeBranch escapes a branch template node: "if", "range
400 func (e *escaper) escapeBranch(c context, n *parse.BranchNode
401     c0 := e.escapeList(c, n.List)
402     if nodeName == "range" && c0.state != stateError {
403         // The "true" branch of a "range" node can e
404         // We check that executing n.List once resul
405         // as executing n.List twice.
406         c1, _ := e.escapeListConditionally(c0, n.Lis
407         c0 = join(c0, c1, n.Line, nodeName)
408         if c0.state == stateError {
409             // Make clear that this is a problem
410             // since developers tend to overlook
411             // debugging templates.
412             c0.err.Line = n.Line
413             c0.err.Description = "on range loop
414         return c0
415     }
416 }
417     c1 := e.escapeList(c, n.ElseList)
418     return join(c0, c1, n.Line, nodeName)
419 }
420
421 // escapeList escapes a list template node.
422 func (e *escaper) escapeList(c context, n *parse.ListNode) c
423     if n == nil {
424         return c
425     }
426     for _, m := range n.Nodes {
427         c = e.escape(c, m)
428     }
429     return c
430 }
431
432 // escapeListConditionally escapes a list node but only pres
433 // inferences in e if the inferences and output context sati
434 // It returns the best guess at an output context, and the r
435 // which is the same as whether e was updated.
436 func (e *escaper) escapeListConditionally(c context, n *pars

```

```

437     e1 := newEscaper(e.tmpl)
438     // Make type inferences available to f.
439     for k, v := range e.output {
440         e1.output[k] = v
441     }
442     c = e1.escapeList(c, n)
443     ok := filter != nil && filter(e1, c)
444     if ok {
445         // Copy inferences and edits from e1 back in
446         for k, v := range e1.output {
447             e.output[k] = v
448         }
449         for k, v := range e1.derived {
450             e.derived[k] = v
451         }
452         for k, v := range e1.called {
453             e.called[k] = v
454         }
455         for k, v := range e1.actionNodeEdits {
456             e.editActionNode(k, v)
457         }
458         for k, v := range e1.templateNodeEdits {
459             e.editTemplateNode(k, v)
460         }
461         for k, v := range e1.textNodeEdits {
462             e.editTextNode(k, v)
463         }
464     }
465     return c, ok
466 }
467
468 // escapeTemplate escapes a {{template}} call node.
469 func (e *escaper) escapeTemplate(c context, n *parse.Templat
470     c, name := e.escapeTree(c, n.Name, n.Line)
471     if name != n.Name {
472         e.editTemplateNode(n, name)
473     }
474     return c
475 }
476
477 // escapeTree escapes the named template starting in the giv
478 // necessary and returns its output context.
479 func (e *escaper) escapeTree(c context, name string, line in
480     // Mangle the template name with the input context t
481     // identifier.
482     dname := c.mangle(name)
483     e.called[dname] = true
484     if out, ok := e.output[dname]; ok {
485         // Already escaped.

```

```

486         return out, dname
487     }
488     t := e.template(name)
489     if t == nil {
490         // Two cases: The template exists but is emp
491         // all. Distinguish the cases in the error m
492         if e.tmpl.set[name] != nil {
493             return context{
494                 state: stateError,
495                 err:   errorf(ErrNoSuchTempl
496                     }, dname
497             }
498             return context{
499                 state: stateError,
500                 err:   errorf(ErrNoSuchTemplate, lin
501                     }, dname
502             }
503         if dname != name {
504             // Use any template derived during an earlie
505             // with different top level templates, or cl
506             dt := e.template(dname)
507             if dt == nil {
508                 dt = template.New(dname)
509                 dt.Tree = &parse.Tree{Name: dname, R
510                     e.derived[dname] = dt
511             }
512             t = dt
513         }
514         return e.computeOutCtx(c, t), dname
515     }
516
517 // computeOutCtx takes a template and its start context and
518 // context while storing any inferences in e.
519 func (e *escaper) computeOutCtx(c context, t *template.Templ
520 // Propagate context over the body.
521 c1, ok := e.escapeTemplateBody(c, t)
522 if !ok {
523     // Look for a fixed point by assuming c1 as
524     if c2, ok2 := e.escapeTemplateBody(c1, t); o
525         c1, ok = c2, true
526     }
527     // Use c1 as the error context if neither as
528 }
529 if !ok && c1.state != stateError {
530     return context{
531         state: stateError,
532         // TODO: Find the first node with a
533         err: errorf(ErrOutputContext, 0, "ca
534     }
535 }

```

```

536         return c1
537     }
538
539     // escapeTemplateBody escapes the given template assuming th
540     // context, and returns the best guess at the output context
541     // assumption was correct.
542     func (e *escaper) escapeTemplateBody(c context, t *template.
543         filter := func(e1 *escaper, c1 context) bool {
544             if c1.state == stateError {
545                 // Do not update the input escaper,
546                 return false
547             }
548             if !e1.called[t.Name()] {
549                 // If t is not recursively called, t
550                 // accurate output context.
551                 return true
552             }
553             // c1 is accurate if it matches our assumed
554             return c.eq(c1)
555         }
556         // We need to assume an output context so that recur
557         // take the fast path out of escapeTree instead of i
558         // Naively assuming that the input context is the sa
559         // works >90% of the time.
560         e.output[t.Name()] = c
561         return e.escapeListConditionally(c, t.Tree.Root, fil
562     }
563
564     // delimEnds maps each delim to a string of characters that
565     var delimEnds = [...]string{
566         delimDoubleQuote: `\"`,
567         delimSingleQuote: "'",
568         // Determined empirically by running the below in va
569         // var div = document.createElement("DIV");
570         // for (var i = 0; i < 0x10000; ++i) {
571         //     div.innerHTML = "<span title=x" + String.fromCh
572         //     if (div.getElementsByTagName("SPAN")[0].title.i
573         //         document.write("<p>U+" + i.toString(16));
574         // }
575         delimSpaceOrTagEnd: " \t\n\f\r>",
576     }
577
578     var doctypeBytes = []byte("<!DOCTYPE")
579
580     // escapeText escapes a text template node.
581     func (e *escaper) escapeText(c context, n *parse.TextNode) c
582         s, written, i, b := n.Text, 0, 0, new(bytes.Buffer)
583         for i != len(s) {
584             c1, nread := contextAfterText(c, s[i:])

```

```

585         i1 := i + nread
586         if c.state == stateText || c.state == stateR
587             end := i1
588             if c1.state != c.state {
589                 for j := end - 1; j >= i; j-
590                     if s[j] == '<' {
591                         end = j
592                         break
593                     }
594             }
595         }
596         for j := i; j < end; j++ {
597             if s[j] == '<' && !bytes.Has
598                 b.Write(s[written:j]
599                 b.WriteString("<";
600                 written = j + 1
601             }
602         }
603     } else if isComment(c.state) && c.delim == d
604         switch c.state {
605             case stateJSBlockCmt:
606                 // http://es5.github.com/#x7
607                 // "Comments behave like whi
608                 // discarded except that, if
609                 // contains a line terminato
610                 // the entire comment is con
611                 // LineTerminator for purpos
612                 // the syntactic grammar."
613                 if bytes.IndexAny(s[written:
614                     b.WriteByte('\n')
615                 } else {
616                     b.WriteByte(' ')
617                 }
618             case stateCSSBlockCmt:
619                 b.WriteByte(' ')
620         }
621         written = i1
622     }
623     if c.state != c1.state && isComment(c1.state
624         // Preserve the portion between writ
625         cs := i1 - 2
626         if c1.state == stateHTMLCmt {
627             // "<!--" instead of "/*" or
628             cs -= 2
629         }
630         b.Write(s[written:cs])
631         written = i1
632     }
633     if i == i1 && c.state == c1.state {

```

```

634             panic(fmt.Sprintf("infinite loop fro
635         }
636         c, i = c1, i1
637     }
638
639     if written != 0 && c.state != stateError {
640         if !isComment(c.state) || c.delim != delimNo
641             b.Write(n.Text[written:])
642         }
643         e.editTextNode(n, b.Bytes())
644     }
645     return c
646 }
647
648 // contextAfterText starts in context c, consumes some token
649 // s, then returns the context after those tokens and the un
650 func contextAfterText(c context, s []byte) (context, int) {
651     if c.delim == delimNone {
652         c1, i := tSpecialTagEnd(c, s)
653         if i == 0 {
654             // A special end tag (`</script>`) h
655             // all content preceding it has been
656             return c1, 0
657         }
658         // Consider all content up to any end tag.
659         return transitionFunc[c.state](c, s[:i])
660     }
661
662     i := bytes.IndexAny(s, delimEnds[c.delim])
663     if i == -1 {
664         i = len(s)
665     }
666     if c.delim == delimSpaceOrTagEnd {
667         // http://www.w3.org/TR/html5/tokenization.h
668         // lists the runes below as error characters
669         // Error out because HTML parsers may differ
670         // "<a id= onclick=f(" ends inside id's
671         // "<a class=`foo " ends inside a val
672         // "<a style=font:'Arial'" needs open-quote
673         // IE treats `` as a quotation character.
674         if j := bytes.IndexAny(s[:i], "\\\"'<="); j >
675             return context{
676                 state: stateError,
677                 err:   errorf(ErrBadHTML, 0,
678                     }, len(s)
679             }
680     }
681     if i == len(s) {
682         // Remain inside the attribute.
683         // Decode the value so non-HTML rules can ea

```

```

684             //      <button onclick="alert(&quot;Hi!&quot;
685             // without having to entity decode token bou
686             for u := []byte(html.UnescapeString(string(s
687                 c1, i1 := transitionFunc[c.state](c,
688                 c, u = c1, u[i1:]
689             }
690             return c, len(s)
691         }
692         if c.delim != delimSpaceOrTagEnd {
693             // Consume any quote.
694             i++
695         }
696         // On exiting an attribute, we discard all state inf
697         // except the state and element.
698         return context{state: stateTag, element: c.element},
699     }
700
701 // editActionNode records a change to an action pipeline for
702 func (e *escaper) editActionNode(n *parse.ActionNode, cmds []
703     if _, ok := e.actionNodeEdits[n]; ok {
704         panic(fmt.Sprintf("node %s shared between te
705     }
706     e.actionNodeEdits[n] = cmds
707 }
708
709 // editTemplateNode records a change to a {{template}} calle
710 func (e *escaper) editTemplateNode(n *parse.TemplateNode, ca
711     if _, ok := e.templateNodeEdits[n]; ok {
712         panic(fmt.Sprintf("node %s shared between te
713     }
714     e.templateNodeEdits[n] = callee
715 }
716
717 // editTextNode records a change to a text node for later co
718 func (e *escaper) editTextNode(n *parse.TextNode, text []byt
719     if _, ok := e.textNodeEdits[n]; ok {
720         panic(fmt.Sprintf("node %s shared between te
721     }
722     e.textNodeEdits[n] = text
723 }
724
725 // commit applies changes to actions and template calls need
726 // autoescape content and adds any derived templates to the
727 func (e *escaper) commit() {
728     for name := range e.output {
729         e.template(name).Funcs(funcMap)
730     }
731     for _, t := range e.derived {
732         if _, err := e.tmpl.text.AddParseTree(t.Name

```

```

733             panic("error adding derived template
734         }
735     }
736     for n, s := range e.actionNodeEdits {
737         ensurePipelineContains(n.Pipe, s)
738     }
739     for n, name := range e.templateNodeEdits {
740         n.Name = name
741     }
742     for n, s := range e.textNodeEdits {
743         n.Text = s
744     }
745 }
746
747 // template returns the named template given a mangled templ
748 func (e *escaper) template(name string) *template.Template {
749     t := e.tpl.text.Lookup(name)
750     if t == nil {
751         t = e.derived[name]
752     }
753     return t
754 }
755
756 // Forwarding functions so that clients need only import thi
757 // to reach the general escaping functions of text/template.
758
759 // HTML_ESCAPE writes to w the escaped HTML equivalent of the
760 func HTML_ESCAPE(w io.Writer, b []byte) {
761     template.HTML_ESCAPE(w, b)
762 }
763
764 // HTML_ESCAPE_STRING returns the escaped HTML equivalent of t
765 func HTML_ESCAPE_STRING(s string) string {
766     return template.HTML_ESCAPE_STRING(s)
767 }
768
769 // HTML_ESCAPER returns the escaped HTML equivalent of the te
770 // representation of its arguments.
771 func HTML_ESCAPER(args ...interface{}) string {
772     return template.HTML_ESCAPER(args...)
773 }
774
775 // JSESCAPE writes to w the escaped JavaScript equivalent of
776 func JSESCAPE(w io.Writer, b []byte) {
777     template.JSESCAPE(w, b)
778 }
779
780 // JSESCAPE_STRING returns the escaped JavaScript equivalent
781 func JSESCAPE_STRING(s string) string {

```

```
782         return template.JSEscapeString(s)
783     }
784
785     // JSEscaper returns the escaped JavaScript equivalent of th
786     // representation of its arguments.
787     func JSEscaper(args ...interface{}) string {
788         return template.JSEscaper(args...)
789     }
790
791     // URLQueryEscaper returns the escaped value of the textual
792     // its arguments in a form suitable for embedding in a URL q
793     func URLQueryEscaper(args ...interface{}) string {
794         return template.URLQueryEscaper(args...)
795     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/html.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "bytes"
9     "fmt"
10    "strings"
11    "unicode/utf8"
12 )
13
14 // htmlNospaceEscaper escapes for inclusion in unquoted attr
15 func htmlNospaceEscaper(args ...interface{}) string {
16     s, t := stringify(args...)
17     if t == contentTypeHTML {
18         return htmlReplacer(stripTags(s), htmlNospac
19     }
20     return htmlReplacer(s, htmlNospaceReplacementTable,
21 }
22
23 // attrEscaper escapes for inclusion in quoted attribute val
24 func attrEscaper(args ...interface{}) string {
25     s, t := stringify(args...)
26     if t == contentTypeHTML {
27         return htmlReplacer(stripTags(s), htmlNormRe
28     }
29     return htmlReplacer(s, htmlReplacementTable, true)
30 }
31
32 // rCDATAEscaper escapes for inclusion in an RCDATA element
33 func rCDATAEscaper(args ...interface{}) string {
34     s, t := stringify(args...)
35     if t == contentTypeHTML {
36         return htmlReplacer(s, htmlNormReplacementTa
37     }
38     return htmlReplacer(s, htmlReplacementTable, true)
39 }
40
41 // htmlEscaper escapes for inclusion in HTML text.
```

```

42 func htmlEscaper(args ...interface{}) string {
43     s, t := stringify(args...)
44     if t == contentTypeHTML {
45         return s
46     }
47     return htmlReplacer(s, htmlReplacementTable, true)
48 }
49
50 // htmlReplacementTable contains the runes that need to be e
51 // inside a quoted attribute value or in a text node.
52 var htmlReplacementTable = []string{
53     // http://www.w3.org/TR/html5/tokenization.html#attr
54     // U+0000 NULL Parse error. Append a U+FFFD REPLACEM
55     // CHARACTER character to the current attribute's va
56     // "
57     // and similarly
58     // http://www.w3.org/TR/html5/tokenization.html#befo
59     0:    "\uFFFD",
60     '"' : "&#34;",
61     '&' : "&amp;",
62     '\'' : "&#39;",
63     '+' : "&#43;",
64     '<' : "&lt;",
65     '>' : "&gt;",
66 }
67
68 // htmlNormReplacementTable is like htmlReplacementTable but
69 // avoid over-encoding existing entities.
70 var htmlNormReplacementTable = []string{
71     0:    "\uFFFD",
72     '"' : "&#34;",
73     '\'' : "&#39;",
74     '+' : "&#43;",
75     '<' : "&lt;",
76     '>' : "&gt;",
77 }
78
79 // htmlNospaceReplacementTable contains the runes that need
80 // inside an unquoted attribute value.
81 // The set of runes escaped is the union of the HTML special
82 // those determined by running the JS below in browsers:
83 // <div id=d></div>
84 // <script>(function () {
85 // var a = [], d = document.getElementById("d"), i, c, s;
86 // for (i = 0; i < 0x10000; ++i) {
87 //   c = String.fromCharCode(i);
88 //   d.innerHTML = "<span title=" + c + "lt" + c + "></span>";
89 //   s = d.getElementsByTagName("SPAN")[0];
90 //   if (!s || s.title !== c + "lt" + c) { a.push(i.toString
91 // }

```

```

92 // document.write(a.join(", "));
93 // }())</script>
94 var htmlNospaceReplacementTable = []string{
95     0:    "�",
96     '\t': "#9;",
97     '\n': "#10;",
98     '\v': "#11;",
99     '\f': "#12;",
100    '\r': "#13;",
101    ' ': "#32;",
102    '"': "#34;",
103    '&': "#39;",
104    '\ ': "#39;",
105    '+': "#43;",
106    '<': "#60;",
107    '=': "#61;",
108    '>': "#62;",
109    // A parse error in the attribute value (unquoted) a
110    // before attribute value states.
111    // Treated as a quoting character by IE.
112    '`': "#96;",
113 }
114
115 // htmlNospaceNormReplacementTable is like htmlNospaceReplac
116 // without '&' to avoid over-encoding existing entities.
117 var htmlNospaceNormReplacementTable = []string{
118     0:    "�",
119     '\t': "#9;",
120     '\n': "#10;",
121     '\v': "#11;",
122     '\f': "#12;",
123     '\r': "#13;",
124     ' ': "#32;",
125     '"': "#34;",
126     '\ ': "#39;",
127     '+': "#43;",
128     '<': "#60;",
129     '=': "#61;",
130     '>': "#62;",
131    // A parse error in the attribute value (unquoted) a
132    // before attribute value states.
133    // Treated as a quoting character by IE.
134    '`': "#96;",
135 }
136
137 // htmlReplacer returns s with runes replaced according to r
138 // and when badRunes is true, certain bad runes are allowed
139 func htmlReplacer(s string, replacementTable []string, badRu
140     written, b := 0, new(bytes.Buffer)

```

```

141     for i, r := range s {
142         if int(r) < len(replacementTable) {
143             if repl := replacementTable[r]; len(
144                 b.WriteString(s[written:i])
145                 b.WriteString(repl)
146                 // Valid as long as replacem
147                 // include anything above 0x
148                 written = i + utf8.RuneLen(r)
149             }
150         } else if badRunes {
151             // No-op.
152             // IE does not allow these ranges in
153         } else if 0xfdd0 <= r && r <= 0xfdef || 0xff
154             fmt.Fprintf(b, "%s&#x%x;", s[written
155             written = i + utf8.RuneLen(r)
156         }
157     }
158     if written == 0 {
159         return s
160     }
161     b.WriteString(s[written:])
162     return b.String()
163 }
164
165 // stripTags takes a snippet of HTML and returns only the te
166 // For example, `&iexcl;Hi!</b> <script>...</script>` ->
167 func stripTags(html string) string {
168     var b bytes.Buffer
169     s, c, i, allText := []byte(html), context{}, 0, true
170     // Using the transition funcs helps us avoid manglin
171     // `
```

```

190                                     }
191                                     }
192                                     b.Write(s[i:j])
193                                 } else {
194                                     allText = false
195                                 }
196                                 c, i = d, i1
197                                 continue
198                             }
199                             i1 := i + bytes.IndexAny(s[i:], delimEnds[c.
200                             if i1 < i {
201                                 break
202                             }
203                             if c.delim != delimSpaceOrTagEnd {
204                                 // Consume any quote.
205                                 i1++
206                             }
207                             c, i = context{state: stateTag, element: c.e
208                         }
209                         if allText {
210                             return html
211                         } else if c.state == stateText || c.state == stateRC
212                             b.Write(s[i:])
213                     }
214                     return b.String()
215 }
216
217 // htmlNameFilter accepts valid parts of an HTML attribute o
218 // a known-safe HTML attribute.
219 func htmlNameFilter(args ...interface{}) string {
220     s, t := stringify(args...)
221     if t == contentTypeHTMLAttr {
222         return s
223     }
224     if len(s) == 0 {
225         // Avoid violation of structure preservation
226         // <input checked {{.K}}={{.V}}>.
227         // Without this, if .K is empty then .V is t
228         // checked, but otherwise .V is the value of
229         // named .K.
230         return filterFailsafe
231     }
232     s = strings.ToLower(s)
233     if t := attrType(s); t != contentTypePlain {
234         // TODO: Split attr and element name part fi
235         // attributes.
236         return filterFailsafe
237     }
238     for _, r := range s {
239         switch {

```

```

240             case '0' <= r && r <= '9':
241             case 'a' <= r && r <= 'z':
242             default:
243                 return filterFailsafe
244             }
245     }
246     return s
247 }
248
249 // commentEscaper returns the empty string regardless of inp
250 // Comment content does not correspond to any parsed structu
251 // human-readable content, so the simplest and most secure p
252 // content interpolated into comments.
253 // This approach is equally valid whether or not static comm
254 // removed from the template.
255 func commentEscaper(args ...interface{}) string {
256     return ""
257 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/js.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "bytes"
9     "encoding/json"
10    "fmt"
11    "reflect"
12    "strings"
13    "unicode/utf8"
14 )
15
16 // nextJSCtx returns the context that determines whether a s
17 // given run of tokens tokens starts a regular expression in
18 // operator: / or /=.
19 //
20 // This assumes that the token run does not include any stri
21 // tokens, regular expression literal tokens, or division op
22 //
23 // This fails on some valid but nonsensical JavaScript progr
24 // "x = ++/foo/i" which is quite different than "x++/foo/i",
25 // fail on any known useful programs. It is based on the dra
26 // JavaScript 2.0 lexical grammar and requires one token of
27 // http://www.mozilla.org/js/language/js20-2000-07/rationale
28 func nextJSCtx(s []byte, preceding jsCtx) jsCtx {
29     s = bytes.TrimRight(s, "\t\n\f\r \u2028\u2029")
30     if len(s) == 0 {
31         return preceding
32     }
33
34     // All cases below are in the single-byte UTF-8 grou
35     switch c, n := s[len(s)-1], len(s); c {
36     case '+', '-':
37         // ++ and -- are not regexp preceders, but +
38         // they are used as infix or prefix operator
39         start := n - 1
40         // Count the number of adjacent dashes or pl
41         for start > 0 && s[start-1] == c {
```

```

42         start--
43     }
44     if (n-start)&1 == 1 {
45         // Reached for trailing minus signs
46         // same as "-- -".
47         return jsCtxRegexp
48     }
49     return jsCtxDivOp
50 case '.':
51     // Handle "42."
52     if n != 1 && '0' <= s[n-2] && s[n-2] <= '9'
53         return jsCtxDivOp
54     }
55     return jsCtxRegexp
56 // Suffixes for all punctuators from section 7.7 of
57 // that only end binary operators not handled above.
58 case ',', '<', '>', '=', '*', '%', '&', '|', '^', '?'
59     return jsCtxRegexp
60 // Suffixes for all punctuators from section 7.7 of
61 // that are prefix operators not handled above.
62 case '!', '~':
63     return jsCtxRegexp
64 // Matches all the punctuators from section 7.7 of t
65 // that are open brackets not handled above.
66 case '(', '[':
67     return jsCtxRegexp
68 // Matches all the punctuators from section 7.7 of t
69 // that precede expression starts.
70 case ':', ';', '{':
71     return jsCtxRegexp
72 // CAVEAT: the close punctuators ('}', ']', ')') pre
73 // are handled in the default except for '}' which c
74 // division op as in
75 //     ({ valueOf: function () { return 42 } } / 2
76 // which is valid, but, in practice, developers don'
77 // literals, so our heuristic works well for code li
78 //     function () { ... } /foo/.test(x) && sideEffe
79 // The ')' punctuator can precede a regular expressi
80 //     if (b) /foo/.test(x) && ...
81 // but this is much less likely than
82 //     (a + b) / c
83 case '}':
84     return jsCtxRegexp
85 default:
86     // Look for an IdentifierName and see if it
87     // can precede a regular expression.
88     j := n
89     for j > 0 && isJSIdentPart(rune(s[j-1])) {
90         j--
91     }

```

```

92         if regexpPrecederKeywords[string(s[j:])] {
93             return jsCtxRegexp
94         }
95     }
96     // Otherwise is a punctuator not listed above, or
97     // a string which precedes a div op, or an identifie
98     // which precedes a div op.
99     return jsCtxDivOp
100 }
101
102 // regexpPrecederKeywords is a set of reserved JS keywords th
103 // regular expression in JS source.
104 var regexpPrecederKeywords = map[string]bool{
105     "break":      true,
106     "case":       true,
107     "continue":   true,
108     "delete":     true,
109     "do":         true,
110     "else":       true,
111     "finally":    true,
112     "in":         true,
113     "instanceof": true,
114     "return":     true,
115     "throw":     true,
116     "try":       true,
117     "typeof":    true,
118     "void":      true,
119 }
120
121 var jsonMarshalType = reflect.TypeOf((*json.Marshaler)(nil))
122
123 // indirectToJSONMarshaler returns the value, after derefere
124 // as necessary to reach the base type (or nil) or an implem
125 func indirectToJSONMarshaler(a interface{}) interface{} {
126     v := reflect.ValueOf(a)
127     for !v.Type().Implements(jsonMarshalType) && v.Kind(
128         v = v.Elem()
129     }
130     return v.Interface()
131 }
132
133 // jsValEscaper escapes its inputs to a JS Expression (secti
134 // neither side-effects nor free variables outside (NaN, Inf
135 func jsValEscaper(args ...interface{}) string {
136     var a interface{}
137     if len(args) == 1 {
138         a = indirectToJSONMarshaler(args[0])
139         switch t := a.(type) {
140         case JS:

```

```

141         return string(t)
142     case JSStr:
143         // TODO: normalize quotes.
144         return `"` + string(t) + "`
145     case json.Marshaler:
146         // Do not treat as a Stringer.
147     case fmt.Stringer:
148         a = t.String()
149     }
150 } else {
151     for i, arg := range args {
152         args[i] = indirectToJSONMarshaler(arg)
153     }
154     a = fmt.Sprint(args...)
155 }
156 // TODO: detect cycles before calling Marshal which
157 // cyclic data. This may be an unacceptable DoS risk
158
159 b, err := json.Marshal(a)
160 if err != nil {
161     // Put a space before comment so that if it
162     // a division operator it is not turned into
163     // x/{{y}}
164     // turning into
165     // x/* error marshalling y:
166     // second line of error message */n
167     return fmt.Sprintf(" /* %s */null ", strings
168 }
169
170 // TODO: maybe post-process output to prevent it fro
171 // "<!--", "-->", "<![CDATA[", "]]>", or "</script"
172 // in case custommarshallers produce output contain
173
174 // TODO: Maybe abbreviate \u00ab to \xab to produce
175 if len(b) == 0 {
176     // In, `x=y/{{.}}*z` a json.Marshaler that p
177     // not cause the output `x=y/*z`.
178     return " null "
179 }
180 first, _ := utf8.DecodeRune(b)
181 last, _ := utf8.DecodeLastRune(b)
182 var buf bytes.Buffer
183 // Prevent IdentifierNames and NumericLiterals from
184 // keywords: in, instanceof, typeof, void
185 pad := isJSIdentPart(first) || isJSIdentPart(last)
186 if pad {
187     buf.WriteByte(' ')
188 }
189 written := 0

```

```

190         // Make sure that json.Marshal escapes codepoints U+
191         // so it falls within the subset of JSON which is va
192         for i := 0; i < len(b); {
193             rune, n := utf8.DecodeRune(b[i:])
194             repl := ""
195             if rune == 0x2028 {
196                 repl = `\u2028`
197             } else if rune == 0x2029 {
198                 repl = `\u2029`
199             }
200             if repl != "" {
201                 buf.Write(b[written:i])
202                 buf.WriteString(repl)
203                 written = i + n
204             }
205             i += n
206         }
207         if buf.Len() != 0 {
208             buf.Write(b[written:])
209             if pad {
210                 buf.WriteByte(' ')
211             }
212             b = buf.Bytes()
213         }
214         return string(b)
215     }
216
217     // jsStrEscaper produces a string that can be included betwe
218     // JavaScript source, in JavaScript embedded in an HTML5 <sc
219     // or in an HTML5 event handler attribute such as onclick.
220     func jsStrEscaper(args ...interface{}) string {
221         s, t := stringify(args...)
222         if t == contentTypeJSStr {
223             return replace(s, jsStrNormReplacementTable)
224         }
225         return replace(s, jsStrReplacementTable)
226     }
227
228     // jsRegexEscaper behaves like jsStrEscaper but escapes reg
229     // specials so the result is treated literally when included
230     // expression literal. /foo{{.X}}bar/ matches the string "fo
231     // the literal text of {{.X}} followed by the string "bar".
232     func jsRegexEscaper(args ...interface{}) string {
233         s, _ := stringify(args...)
234         s = replace(s, jsRegexReplacementTable)
235         if s == "" {
236             // /{{.X}}/ should not produce a line commen
237             return "(?:)"
238         }
239         return s

```

```

240 }
241
242 // replace replaces each rune r of s with replacementTable[r]
243 // r < len(replacementTable). If replacementTable[r] is the
244 // no replacement is made.
245 // It also replaces runes U+2028 and U+2029 with the raw str
246 // ``\u2028``.
247 func replace(s string, replacementTable []string) string {
248     var b bytes.Buffer
249     written := 0
250     for i, r := range s {
251         var repl string
252         switch {
253             case int(r) < len(replacementTable) && repla
254                 repl = replacementTable[r]
255             case r == ``\u2028``:
256                 repl = ``\u2028``
257             case r == ``\u2029``:
258                 repl = ``\u2029``
259             default:
260                 continue
261         }
262         b.WriteString(s[written:i])
263         b.WriteString(repl)
264         written = i + utf8.RuneLen(r)
265     }
266     if written == 0 {
267         return s
268     }
269     b.WriteString(s[written:])
270     return b.String()
271 }
272
273 var jsStrReplacementTable = []string{
274     0: ``\0``,
275     '\t': ``\t``,
276     '\n': ``\n``,
277     '\v': ``\x0b``, // ``\v`` == ``v`` on IE 6.
278     '\f': ``\f``,
279     '\r': ``\r``,
280     // Encode HTML specials as hex so the output can be
281     // in HTML attributes without further encoding.
282     '"': ``\x22``,
283     '&': ``\x26``,
284     '\': ``\x27``,
285     '+': ``\x2b``,
286     '/': ``\/``,
287     '<': ``\x3c``,
288     '>': ``\x3e``,

```

```

289     '\\': '\\`,
290 }
291
292 // jsStrNormReplacementTable is like jsStrReplacementTable b
293 // overencode existing escapes since this table has no entry
294 var jsStrNormReplacementTable = []string{
295     0:    '\0`,
296     '\t': '\t`,
297     '\n': '\n`,
298     '\v': '\x0b`, // "\v" == "v" on IE 6.
299     '\f': '\f`,
300     '\r': '\r`,
301     // Encode HTML specials as hex so the output can be
302     // in HTML attributes without further encoding.
303     '"':  '\x22`,
304     '&':  '\x26`,
305     '\':  '\x27`,
306     '+':  '\x2b`,
307     '/':  '\x2f`,
308     '<':  '\x3c`,
309     '>':  '\x3e`,
310 }
311
312 var jsRegexReplacementTable = []string{
313     0:    '\0`,
314     '\t': '\t`,
315     '\n': '\n`,
316     '\v': '\x0b`, // "\v" == "v" on IE 6.
317     '\f': '\f`,
318     '\r': '\r`,
319     // Encode HTML specials as hex so the output can be
320     // in HTML attributes without further encoding.
321     '"':  '\x22`,
322     '$':  '\x24`,
323     '&':  '\x26`,
324     '\':  '\x27`,
325     '(' : '\x28`,
326     ')' : '\x29`,
327     '*' : '\x2a`,
328     '+' : '\x2b`,
329     '-' : '\x2d`,
330     '.' : '\x2e`,
331     '/' : '\x2f`,
332     '<':  '\x3c`,
333     '>':  '\x3e`,
334     '?' : '\x3f`,
335     '[' : '\x5b`,
336     '\\': '\\`,
337     ']' : '\x5d`,

```

```

338         '^':  `\\^`,
339         '{':  `\\{`,
340         '|':  `\\|`,
341         '}'': `\\}`,
342     }
343
344     // isJSIdentPart returns whether the given rune is a JS iden
345     // It does not handle all the non-Latin letters, joiners, an
346     // but it does handle every codepoint that can occur in a nu
347     // a keyword.
348     func isJSIdentPart(r rune) bool {
349         switch {
350         case r == '$':
351             return true
352         case '0' <= r && r <= '9':
353             return true
354         case 'A' <= r && r <= 'Z':
355             return true
356         case r == '_':
357             return true
358         case 'a' <= r && r <= 'z':
359             return true
360         }
361         return false
362     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/template.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "fmt"
9     "io"
10    "io/ioutil"
11    "path/filepath"
12    "sync"
13    "text/template"
14    "text/template/parse"
15 )
16
17 // Template is a specialized template.Template that produces
18 // document fragment.
19 type Template struct {
20     escaped bool
21     // We could embed the text/template field, but it's
22     // we need to keep our version of the name space and
23     // template's in sync.
24     text      *template.Template
25     *nameSpace // common to all associated templates
26 }
27
28 // nameSpace is the data structure shared by all templates i
29 type nameSpace struct {
30     mu sync.Mutex
31     set map[string]*Template
32 }
33
34 // Templates returns a slice of the templates associated wit
35 // itself.
36 func (t *Template) Templates() []*Template {
37     ns := t.nameSpace
38     ns.mu.Lock()
39     defer ns.mu.Unlock()
40     // Return a slice so we don't expose the map.
41     m := make([]*Template, 0, len(ns.set))
```

```

42         for _, v := range ns.set {
43             m = append(m, v)
44         }
45         return m
46     }
47
48     // Execute applies a parsed template to the specified data o
49     // writing the output to wr.
50     func (t *Template) Execute(wr io.Writer, data interface{}) (
51         t.nameSpace.mu.Lock()
52         if !t.escaped {
53             if err = escapeTemplates(t, t.Name()); err !=
54                 t.escaped = true
55         }
56     }
57     t.nameSpace.mu.Unlock()
58     if err != nil {
59         return
60     }
61     return t.text.Execute(wr, data)
62 }
63
64 // ExecuteTemplate applies the template associated with t th
65 // name to the specified data object and writes the output t
66 func (t *Template) ExecuteTemplate(wr io.Writer, name string
67     tpl, err := t.lookupAndEscapeTemplate(name)
68     if err != nil {
69         return err
70     }
71     return tpl.text.Execute(wr, data)
72 }
73
74 // lookupAndEscapeTemplate guarantees that the template with
75 // is escaped, or returns an error if it cannot be. It retur
76 // template.
77 func (t *Template) lookupAndEscapeTemplate(name string) (tmp
78     t.nameSpace.mu.Lock()
79     defer t.nameSpace.mu.Unlock()
80     tpl = t.set[name]
81     if tpl == nil {
82         return nil, fmt.Errorf("html/template: %q is
83     }
84     if tpl.text.Tree == nil || tpl.text.Root == nil {
85         return nil, fmt.Errorf("html/template: %q is
86     }
87     if t.text.Lookup(name) == nil {
88         panic("html/template internal error: templat
89     }
90     if tpl != nil && !tpl.escaped {
91         err = escapeTemplates(tpl, name)

```

```

92         }
93         return tmpl, err
94     }
95
96     // Parse parses a string into a template. Nested template de
97     // will be associated with the top-level template t. Parse r
98     // called multiple times to parse definitions of templates t
99     // with t. It is an error if a resulting template is non-emp
100    // content other than template definitions) and would replac
101    // non-empty template with the same name. (In multiple call
102    // with the same receiver template, only one call can contai
103    // other than space, comments, and template definitions.)
104    func (t *Template) Parse(src string) (*Template, error) {
105        t.nameSpace.mu.Lock()
106        t.escaped = false
107        t.nameSpace.mu.Unlock()
108        ret, err := t.text.Parse(src)
109        if err != nil {
110            return nil, err
111        }
112        // In general, all the named templates might have ch
113        // Regardless, some new ones may have been defined.
114        // The template.Template set has been updated; updat
115        t.nameSpace.mu.Lock()
116        defer t.nameSpace.mu.Unlock()
117        for _, v := range ret.Templates() {
118            name := v.Name()
119            tmpl := t.set[name]
120            if tmpl == nil {
121                tmpl = t.new(name)
122            }
123            tmpl.escaped = false
124            tmpl.text = v
125        }
126        return t, nil
127    }
128
129    // AddParseTree creates a new template with the name and par
130    // and associates it with t.
131    //
132    // It returns an error if t has already been executed.
133    func (t *Template) AddParseTree(name string, tree *parse.Tre
134        t.nameSpace.mu.Lock()
135        defer t.nameSpace.mu.Unlock()
136        if t.escaped {
137            return nil, fmt.Errorf("html/template: canno
138        }
139        text, err := t.text.AddParseTree(name, tree)
140        if err != nil {

```

```

141         return nil, err
142     }
143     ret := &Template{
144         false,
145         text,
146         t.nameSpace,
147     }
148     t.set[name] = ret
149     return ret, nil
150 }
151
152 // Clone returns a duplicate of the template, including all
153 // templates. The actual representation is not copied, but t
154 // associated templates is, so further calls to Parse in the
155 // templates to the copy but not to the original. Clone can
156 // common templates and use them with variant definitions fo
157 // by adding the variants after the clone is made.
158 //
159 // It returns an error if t has already been executed.
160 func (t *Template) Clone() (*Template, error) {
161     t.nameSpace.mu.Lock()
162     defer t.nameSpace.mu.Unlock()
163     if t.escaped {
164         return nil, fmt.Errorf("html/template: canno
165     }
166     textClone, err := t.text.Clone()
167     if err != nil {
168         return nil, err
169     }
170     ret := &Template{
171         false,
172         textClone,
173         &nameSpace{
174             set: make(map[string]*Template),
175         },
176     }
177     for _, x := range textClone.Templates() {
178         name := x.Name()
179         src := t.set[name]
180         if src == nil || src.escaped {
181             return nil, fmt.Errorf("html/templat
182         }
183         if x.Tree != nil {
184             x.Tree = &parse.Tree{
185                 Name: x.Tree.Name,
186                 Root: x.Tree.Root.CopyList()
187             }
188         }
189         ret.set[name] = &Template{

```

```

190             false,
191             x,
192             ret.nameSpace,
193         }
194     }
195     return ret, nil
196 }
197
198 // New allocates a new HTML template with the given name.
199 func New(name string) *Template {
200     tmpl := &Template{
201         false,
202         template.New(name),
203         &nameSpace{
204             set: make(map[string]*Template),
205         },
206     }
207     tmpl.set[name] = tmpl
208     return tmpl
209 }
210
211 // New allocates a new HTML template associated with the giv
212 // and with the same delimiters. The association, which is t
213 // allows one template to invoke another with a {{template}}
214 func (t *Template) New(name string) *Template {
215     t.nameSpace.mu.Lock()
216     defer t.nameSpace.mu.Unlock()
217     return t.new(name)
218 }
219
220 // new is the implementation of New, without the lock.
221 func (t *Template) new(name string) *Template {
222     tmpl := &Template{
223         false,
224         t.text.New(name),
225         t.nameSpace,
226     }
227     tmpl.set[name] = tmpl
228     return tmpl
229 }
230
231 // Name returns the name of the template.
232 func (t *Template) Name() string {
233     return t.text.Name()
234 }
235
236 // FuncMap is the type of the map defining the mapping from
237 // functions. Each function must have either a single return
238 // return values of which the second has type error. In that
239 // second (error) argument evaluates to non-nil during execu

```

```

240 // terminates and Execute returns that error. FuncMap has th
241 // as template.FuncMap, copied here so clients need not impo
242 type FuncMap map[string]interface{}
243
244 // Funcs adds the elements of the argument map to the templa
245 // It panics if a value in the map is not a function with ap
246 // type. However, it is legal to overwrite elements of the m
247 // value is the template, so calls can be chained.
248 func (t *Template) Funcs(funcMap FuncMap) *Template {
249     t.text.Funcs(template.FuncMap(funcMap))
250     return t
251 }
252
253 // Delims sets the action delimiters to the specified string
254 // subsequent calls to Parse, ParseFiles, or ParseGlob. Nest
255 // definitions will inherit the settings. An empty delimiter
256 // corresponding default: {{ or }}.
257 // The return value is the template, so calls can be chained
258 func (t *Template) Delims(left, right string) *Template {
259     t.text.Delims(left, right)
260     return t
261 }
262
263 // Lookup returns the template with the given name that is a
264 // or nil if there is no such template.
265 func (t *Template) Lookup(name string) *Template {
266     t.nameSpace.mu.Lock()
267     defer t.nameSpace.mu.Unlock()
268     return t.set[name]
269 }
270
271 // Must panics if err is non-nil in the same way as template
272 func Must(t *Template, err error) *Template {
273     if err != nil {
274         panic(err)
275     }
276     return t
277 }
278
279 // ParseFiles creates a new Template and parses the template
280 // the named files. The returned template's name will have t
281 // (parsed) contents of the first file. There must be at lea
282 // If an error occurs, parsing stops and the returned *Templ
283 func ParseFiles(fileNames ...string) (*Template, error) {
284     return parseFiles(nil, fileNames...)
285 }
286
287 // ParseFiles parses the named files and associates the resu
288 // t. If an error occurs, parsing stops and the returned tem

```

```

289 // otherwise it is t. There must be at least one file.
290 func (t *Template) ParseFiles(filenamees ...string) (*Templat
291     return parseFiles(t, filenamees...)
292 }
293
294 // parseFiles is the helper for the method and function. If
295 // template is nil, it is created from the first file.
296 func parseFiles(t *Template, filenamees ...string) (*Template
297     if len(filenamees) == 0 {
298         // Not really a problem, but be consistent.
299         return nil, fmt.Errorf("html/template: no fi
300     }
301     for _, filename := range filenamees {
302         b, err := ioutil.ReadFile(filename)
303         if err != nil {
304             return nil, err
305         }
306         s := string(b)
307         name := filepath.Base(filename)
308         // First template becomes return value if no
309         // and we use that one for subsequent New ca
310         // all the templates together. Also, if this
311         // as t, this file becomes the contents of t
312         // t, err := New(name).Funcs(xxx).ParseFile
313         // works. Otherwise we create a new template
314         var tmpl *Template
315         if t == nil {
316             t = New(name)
317         }
318         if name == t.Name() {
319             tmpl = t
320         } else {
321             tmpl = t.New(name)
322         }
323         _, err = tmpl.Parse(s)
324         if err != nil {
325             return nil, err
326         }
327     }
328     return t, nil
329 }
330
331 // ParseGlob creates a new Template and parses the template
332 // files identified by the pattern, which must match at leas
333 // returned template will have the (base) name and (parsed)
334 // first file matched by the pattern. ParseGlob is equivalen
335 // ParseFiles with the list of files matched by the pattern.
336 func ParseGlob(pattern string) (*Template, error) {
337     return parseGlob(nil, pattern)

```

```

338 }
339
340 // ParseGlob parses the template definitions in the files id
341 // pattern and associates the resulting templates with t. Th
342 // processed by filepath.Glob and must match at least one fi
343 // equivalent to calling t.ParseFiles with the list of files
344 // pattern.
345 func (t *Template) ParseGlob(pattern string) (*Template, err
346         return parseGlob(t, pattern)
347 }
348
349 // parseGlob is the implementation of the function and metho
350 func parseGlob(t *Template, pattern string) (*Template, erro
351         filenames, err := filepath.Glob(pattern)
352         if err != nil {
353             return nil, err
354         }
355         if len(filenames) == 0 {
356             return nil, fmt.Errorf("html/template: patte
357         }
358         return parseFiles(t, filenames...)
359 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/transition.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "bytes"
9     "strings"
10 )
11
12 // transitionFunc is the array of context transition functions.
13 // A transition function takes a context and template text i
14 // the updated context and the number of bytes consumed from
15 // input.
16 var transitionFunc = [...]func(context, []byte) (context, in
17     stateText:      tText,
18     stateTag:       tTag,
19     stateAttrName:  tAttrName,
20     stateAfterName: tAfterName,
21     stateBeforeValue: tBeforeValue,
22     stateHTMLCmt:   tHTMLCmt,
23     stateRCDATA:    tSpecialTagEnd,
24     stateAttr:      tAttr,
25     stateURL:       tURL,
26     stateJS:        tJS,
27     stateJSDqStr:   tJSDelimited,
28     stateJSSqStr:   tJSDelimited,
29     stateJSRegex:   tJSDelimited,
30     stateJSBlockCmt: tBlockCmt,
31     stateJSLineCmt: tLineCmt,
32     stateCSS:       tCSS,
33     stateCSSDqStr:  tCSSStr,
34     stateCSSSqStr:  tCSSStr,
35     stateCSSDqURL:  tCSSStr,
36     stateCSSSqURL:  tCSSStr,
37     stateCSSURL:    tCSSStr,
38     stateCSSBlockCmt: tBlockCmt,
39     stateCSSLineCmt: tLineCmt,
40     stateError:     tError,
41 }
```

```

42
43 var commentStart = []byte("<!--")
44 var commentEnd = []byte("-->")
45
46 // tText is the context transition function for the text sta
47 func tText(c context, s []byte) (context, int) {
48     k := 0
49     for {
50         i := k + bytes.IndexByte(s[k:], '<')
51         if i < k || i+1 == len(s) {
52             return c, len(s)
53         } else if i+4 <= len(s) && bytes.Equal(comme
54             return context{state: stateHTMLCmt},
55         }
56         i++
57         end := false
58         if s[i] == '/' {
59             if i+1 == len(s) {
60                 return c, len(s)
61             }
62             end, i = true, i+1
63         }
64         j, e := eatTagName(s, i)
65         if j != i {
66             if end {
67                 e = elementNone
68             }
69             // We've found an HTML tag.
70             return context{state: stateTag, elem
71         }
72         k = j
73     }
74     panic("unreachable")
75 }
76
77 var elementContentType = [...]state{
78     elementNone:    stateText,
79     elementScript:  stateJS,
80     elementStyle:   stateCSS,
81     elementTextarea: stateRCDATA,
82     elementTitle:   stateRCDATA,
83 }
84
85 // tTag is the context transition function for the tag state
86 func tTag(c context, s []byte) (context, int) {
87     // Find the attribute name.
88     i := eatWhiteSpace(s, 0)
89     if i == len(s) {
90         return c, len(s)
91     }

```

```

92     if s[i] == '>' {
93         return context{
94             state:  elementType[c.element],
95             element: c.element,
96             }, i + 1
97     }
98     j, err := eatAttrName(s, i)
99     if err != nil {
100         return context{state: stateError, err: err},
101     }
102     state, attr := stateTag, attrNone
103     if i == j {
104         return context{
105             state: stateError,
106             err:  errorf(ErrBadHTML, 0, "expect
107             }, len(s)
108         }
109     switch attrType(string(s[i:j])) {
110     case contentTypeURL:
111         attr = attrURL
112     case contentTypeCSS:
113         attr = attrStyle
114     case contentTypeJS:
115         attr = attrScript
116     }
117     if j == len(s) {
118         state = stateAttrName
119     } else {
120         state = stateAfterName
121     }
122     return context{state: state, element: c.element, attr
123 }
124
125 // tAttrName is the context transition function for stateAttrName
126 func tAttrName(c context, s []byte) (context, int) {
127     i, err := eatAttrName(s, 0)
128     if err != nil {
129         return context{state: stateError, err: err},
130     } else if i != len(s) {
131         c.state = stateAfterName
132     }
133     return c, i
134 }
135
136 // tAfterName is the context transition function for stateAfterName
137 func tAfterName(c context, s []byte) (context, int) {
138     // Look for the start of the value.
139     i := eatWhiteSpace(s, 0)
140     if i == len(s) {

```

```

141         return c, len(s)
142     } else if s[i] != '=' {
143         // Occurs due to tag ending '>', and valuele
144         c.state = stateTag
145         return c, i
146     }
147     c.state = stateBeforeValue
148     // Consume the "=".
149     return c, i + 1
150 }
151
152 var attrStartStates = [...]state{
153     attrNone:    stateAttr,
154     attrScript: stateJS,
155     attrStyle:   stateCSS,
156     attrURL:    stateURL,
157 }
158
159 // tBeforeValue is the context transition function for state
160 func tBeforeValue(c context, s []byte) (context, int) {
161     i := eatWhiteSpace(s, 0)
162     if i == len(s) {
163         return c, len(s)
164     }
165     // Find the attribute delimiter.
166     delim := delimSpaceOrTagEnd
167     switch s[i] {
168     case '\':
169         delim, i = delimSingleQuote, i+1
170     case '"':
171         delim, i = delimDoubleQuote, i+1
172     }
173     c.state, c.delim, c.attr = attrStartStates[c.attr],
174     return c, i
175 }
176
177 // tHTMLCmt is the context transition function for stateHTML
178 func tHTMLCmt(c context, s []byte) (context, int) {
179     if i := bytes.Index(s, commentEnd); i != -1 {
180         return context{}, i + 3
181     }
182     return c, len(s)
183 }
184
185 // specialTagEndMarkers maps element types to the character
186 // case-insensitively signals the end of the special tag bod
187 var specialTagEndMarkers = [...]string{
188     elementScript: "</script",
189     elementStyle:  "</style",

```

```

190         elementTextarea: "</textarea",
191         elementTitle:    "</title",
192     }
193
194     // tSpecialTagEnd is the context transition function for raw
195     // element states.
196     func tSpecialTagEnd(c context, s []byte) (context, int) {
197         if c.element != elementNone {
198             if i := strings.Index(strings.ToLower(string
199                 return context{}, i
200             }
201         }
202         return c, len(s)
203     }
204
205     // tAttr is the context transition function for the attribut
206     func tAttr(c context, s []byte) (context, int) {
207         return c, len(s)
208     }
209
210     // tURL is the context transition function for the URL state
211     func tURL(c context, s []byte) (context, int) {
212         if bytes.IndexAny(s, "#?") >= 0 {
213             c.urlPart = urlPartQueryOrFrag
214         } else if len(s) != eatWhiteSpace(s, 0) && c.urlPart
215             // HTML5 uses "Valid URL potentially surroun
216             // attrs: http://www.w3.org/TR/html5/index.h
217             c.urlPart = urlPartPreQuery
218         }
219         return c, len(s)
220     }
221
222     // tJS is the context transition function for the JS state.
223     func tJS(c context, s []byte) (context, int) {
224         i := bytes.IndexAny(s, `"/`)
225         if i == -1 {
226             // Entire input is non string, comment, rege
227             c.jsCtx = nextJSctx(s, c.jsCtx)
228             return c, len(s)
229         }
230         c.jsCtx = nextJSctx(s[:i], c.jsCtx)
231         switch s[i] {
232         case '"':
233             c.state, c.jsCtx = stateJSDqStr, jsCtxRegexp
234         case '\':
235             c.state, c.jsCtx = stateJSSqStr, jsCtxRegexp
236         case '/':
237             switch {
238             case i+1 < len(s) && s[i+1] == '/':
239                 c.state, i = stateJSLineCmt, i+1

```

```

240         case i+1 < len(s) && s[i+1] == '*':
241             c.state, i = stateJSBlockCmt, i+1
242         case c.jsCtx == jsCtxRegexp:
243             c.state = stateJSRegexp
244         case c.jsCtx == jsCtxDivOp:
245             c.jsCtx = jsCtxRegexp
246         default:
247             return context{
248                 state: stateError,
249                 err:   errorf(ErrSlashAmbig,
250                     }, len(s)
251             }
252     default:
253         panic("unreachable")
254     }
255     return c, i + 1
256 }
257
258 // tJSDelimited is the context transition function for the J
259 // states.
260 func tJSDelimited(c context, s []byte) (context, int) {
261     specials := `\"`
262     switch c.state {
263     case stateJSSqStr:
264         specials = `\'`
265     case stateJSRegexp:
266         specials = `\/[ ]`
267     }
268
269     k, inCharset := 0, false
270     for {
271         i := k + bytes.IndexAny(s[k:], specials)
272         if i < k {
273             break
274         }
275         switch s[i] {
276         case '\\':
277             i++
278             if i == len(s) {
279                 return context{
280                     state: stateError,
281                     err:   errorf(ErrPar
282                         }, len(s)
283                 }
284             }
285         case '[':
286             inCharset = true
287         case ']':
288             inCharset = false
289         default:

```

```

289             // end delimiter
290             if !inCharset {
291                 c.state, c.jsCtx = stateJS,
292                 return c, i + 1
293             }
294         }
295         k = i + 1
296     }
297
298     if inCharset {
299         // This can be fixed by making context riche
300         // into charsets is desired.
301         return context{
302             state: stateError,
303             err:   errorf(ErrPartialCharset, 0,
304             }, len(s)
305         }
306
307         return c, len(s)
308     }
309
310     var blockCommentEnd = []byte("*/")
311
312     // tBlockCmt is the context transition function for /*commen
313     func tBlockCmt(c context, s []byte) (context, int) {
314         i := bytes.Index(s, blockCommentEnd)
315         if i == -1 {
316             return c, len(s)
317         }
318         switch c.state {
319             case stateJSBlockCmt:
320                 c.state = stateJS
321             case stateCSSBlockCmt:
322                 c.state = stateCSS
323             default:
324                 panic(c.state.String())
325         }
326         return c, i + 2
327     }
328
329     // tLineCmt is the context transition function for //comment
330     func tLineCmt(c context, s []byte) (context, int) {
331         var lineTerminators string
332         var endState state
333         switch c.state {
334             case stateJSLineCmt:
335                 lineTerminators, endState = "\n\r\u2028\u202
336             case stateCSSLineCmt:
337                 lineTerminators, endState = "\n\f\r", stateC

```

```

338         // Line comments are not part of any publish
339         // are supported by the 4 major browsers.
340         // This defines line comments as
341         //     LINECOMMENT ::= "//" [^\n\f\d]*
342         // since http://www.w3.org/TR/css3-syntax/#S
343         // newlines:
344         //     nl ::= #xA | #xD #xA | #xD | #xC
345     default:
346         panic(c.state.String())
347     }
348
349     i := bytes.IndexAny(s, lineTerminators)
350     if i == -1 {
351         return c, len(s)
352     }
353     c.state = endState
354     // Per section 7.4 of EcmaScript 5 : http://es5.gith
355     // "However, the LineTerminator at the end of the li
356     // considered to be part of the single-line comment;
357     // recognized separately by the lexical grammar and
358     // of the stream of input elements for the syntactic
359     return c, i
360 }
361
362 // tCSS is the context transition function for the CSS state
363 func tCSS(c context, s []byte) (context, int) {
364     // CSS quoted strings are almost never used except f
365     // (1) URLs as in background: "/foo.png"
366     // (2) Multiword font-names as in font-family: "Time
367     // (3) List separators in content values as in inlin
368     // <style>
369     //     ul.inlineList { list-style: none; padding:0 }
370     //     ul.inlineList > li { display: inline }
371     //     ul.inlineList > li:before { content: ", " }
372     //     ul.inlineList > li:first-child:before { conten
373     // </style>
374     //     <ul class=inlineList><li>One<li>Two<li>Three</
375     // (4) Attribute value selectors as in a[href="http:
376     //
377     // We conservatively treat all strings as URLs, but
378     // allowances to avoid confusion.
379     //
380     // In (1), our conservative assumption is justified.
381     // In (2), valid font names do not contain ':', '?',
382     // conservative assumption is fine since we will nev
383     // urlPartPreQuery.
384     // In (3), our protocol heuristic should not be trip
385     // should not be non-space content after a '?' or '#'
386     // we only %-encode RFC 3986 reserved characters we
387     // In (4), we should URL escape for URL attributes,

```

```

388 // have the attribute name available if our conserva
389 // proves problematic for real code.
390
391 k := 0
392 for {
393     i := k + bytes.IndexAny(s[k:], `('/\`)
394     if i < k {
395         return c, len(s)
396     }
397     switch s[i] {
398     case '(':
399         // Look for url to the left.
400         p := bytes.TrimRight(s[:i], "\t\n\f\
401         if endsWithCSSKeyword(p, "url") {
402             j := len(s) - len(bytes.Trim
403             switch {
404             case j != len(s) && s[j] ==
405                 c.state, j = stateCS
406             case j != len(s) && s[j] ==
407                 c.state, j = stateCS
408             default:
409                 c.state = stateCSSUR
410             }
411             return c, j
412         }
413     case '/':
414         if i+1 < len(s) {
415             switch s[i+1] {
416             case '/':
417                 c.state = stateCSSLi
418                 return c, i + 2
419             case '*':
420                 c.state = stateCSSBl
421                 return c, i + 2
422             }
423         }
424     case '"':
425         c.state = stateCSSDqStr
426         return c, i + 1
427     case '\\':
428         c.state = stateCSSSqStr
429         return c, i + 1
430     }
431     k = i + 1
432 }
433 panic("unreachable")
434 }
435
436 // tCSSStr is the context transition function for the CSS st

```

```

437 func tCSSStr(c context, s []byte) (context, int) {
438     var endAndEsc string
439     switch c.state {
440     case stateCSSDqStr, stateCSSDqURL:
441         endAndEsc = `\"`
442     case stateCSSSqStr, stateCSSSqURL:
443         endAndEsc = `\'`
444     case stateCSSURL:
445         // Unquoted URLs end with a newline or close
446         // The below includes the wc (whitespace cha
447         endAndEsc = "\\t\n\r )"
448     default:
449         panic(c.state.String())
450     }
451
452     k := 0
453     for {
454         i := k + bytes.IndexAny(s[k:], endAndEsc)
455         if i < k {
456             c, nread := tURL(c, decodeCSS(s[k:]))
457             return c, k + nread
458         }
459         if s[i] == `\"` {
460             i++
461             if i == len(s) {
462                 return context{
463                     state: stateError,
464                     err:   fmt.Errorf("ErrPar
465                 }, len(s)
466             }
467         } else {
468             c.state = stateCSS
469             return c, i + 1
470         }
471         c, _ = tURL(c, decodeCSS(s[:i+1]))
472         k = i + 1
473     }
474     panic("unreachable")
475 }
476
477 // tError is the context transition function for the error s
478 func tError(c context, s []byte) (context, int) {
479     return c, len(s)
480 }
481
482 // eatAttrName returns the largest j such that s[i:j] is an
483 // It returns an error if s[i:] does not look like it begins
484 // attribute name, such as encountering a quote mark without
485 // equals sign.

```

```

486 func eatAttrName(s []byte, i int) (int, *Error) {
487     for j := i; j < len(s); j++ {
488         switch s[j] {
489             case ' ', '\t', '\n', '\f', '\r', '=', '>':
490                 return j, nil
491             case '\\', '"', '<':
492                 // These result in a parse warning i
493                 // indicative of serious problems if
494                 // name in a template.
495                 return -1, errorf(ErrBadHTML, 0, "%q
496             default:
497                 // No-op.
498         }
499     }
500     return len(s), nil
501 }
502
503 var elementNameMap = map[string]element{
504     "script":    elementScript,
505     "style":     elementStyle,
506     "textarea": elementTextarea,
507     "title":     elementTitle,
508 }
509
510 // asciiAlpha returns whether c is an ASCII letter.
511 func asciiAlpha(c byte) bool {
512     return 'A' <= c && c <= 'Z' || 'a' <= c && c <= 'z'
513 }
514
515 // asciiAlphaNum returns whether c is an ASCII letter or dig
516 func asciiAlphaNum(c byte) bool {
517     return asciiAlpha(c) || '0' <= c && c <= '9'
518 }
519
520 // eatTagName returns the largest j such that s[i:j] is a ta
521 func eatTagName(s []byte, i int) (int, element) {
522     if i == len(s) || !asciiAlpha(s[i]) {
523         return i, elementNone
524     }
525     j := i + 1
526     for j < len(s) {
527         x := s[j]
528         if asciiAlphaNum(x) {
529             j++
530             continue
531         }
532         // Allow "x-y" or "x:y" but not "x-", "-y",
533         if (x == ':' || x == '-') && j+1 < len(s) &&
534             j += 2
535         continue

```

```

536         }
537         break
538     }
539     return j, elementNameMap[strings.ToLower(string(s[i:
540 }
541
542 // eatWhiteSpace returns the largest j such that s[i:j] is w
543 func eatWhiteSpace(s []byte, i int) int {
544     for j := i; j < len(s); j++ {
545         switch s[j] {
546             case ' ', '\t', '\n', '\f', '\r':
547                 // No-op.
548             default:
549                 return j
550         }
551     }
552     return len(s)
553 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/html/template/url.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "bytes"
9     "fmt"
10    "strings"
11 )
12
13 // urlFilter returns its input unless it contains an unsafe
14 // case it defangs the entire URL.
15 func urlFilter(args ...interface{}) string {
16     s, t := stringify(args...)
17     if t == contentTypeURL {
18         return s
19     }
20     if i := strings.IndexRune(s, ':'); i >= 0 && strings
21         protocol := strings.ToLower(s[:i])
22         if protocol != "http" && protocol != "https"
23             return "#" + filterFailsafe
24     }
25 }
26 return s
27 }
28
29 // urlEscaper produces an output that can be embedded in a U
30 // The output can be embedded in an HTML attribute without f
31 func urlEscaper(args ...interface{}) string {
32     return urlProcessor(false, args...)
33 }
34
35 // urlEscaper normalizes URL content so it can be embedded i
36 // string or parenthesis delimited url(...).
37 // The normalizer does not encode all HTML specials. Specifi
38 // encode '&' so correct embedding in an HTML attribute requ
39 // '&' to '&amp;'.
40 func urlNormalizer(args ...interface{}) string {
41     return urlProcessor(true, args...)
```

```

42 }
43
44 // urlProcessor normalizes (when norm is true) or escapes it
45 // a valid hierarchical or opaque URL part.
46 func urlProcessor(norm bool, args ...interface{}) string {
47     s, t := stringify(args...)
48     if t == contentTypeURL {
49         norm = true
50     }
51     var b bytes.Buffer
52     written := 0
53     // The byte loop below assumes that all URLs use UTF
54     // content-encoding. This is similar to the URI to I
55     // defined in section 3.1 of RFC 3987, and behaves
56     // EcmaScript builtin encodeURIComponent.
57     // It should not cause any misencoding of URLs in pa
58     // Content-type: text/html;charset=UTF-8.
59     for i, n := 0, len(s); i < n; i++ {
60         c := s[i]
61         switch c {
62             // Single quote and parens are sub-delims in
63             // escape them so the output can be embedded
64             // quoted attributes and unquoted CSS url(..
65             // Single quotes are reserved in URLs, but a
66             // the obsolete "mark" rule in an appendix i
67             // so can be safely encoded.
68             case '!', '#', '$', '&', '*', '+', ',', '/',
69                 if norm {
70                     continue
71                 }
72             // Unreserved according to RFC 3986 sec 2.3
73             // "For consistency, percent-encoded octets
74             // ALPHA (%41-%5A and %61-%7A), DIGIT (%30-%
75             // period (%2E), underscore (%5F), or tilde
76             // created by URI producers
77             case '-', '.', '_', '~':
78                 continue
79             case '%':
80                 // When normalizing do not re-encode
81                 if norm && i+2 < len(s) && isHex(s[i
82                     continue
83                 }
84             default:
85                 // Unreserved according to RFC 3986
86                 if 'a' <= c && c <= 'z' {
87                     continue
88                 }
89                 if 'A' <= c && c <= 'Z' {
90                     continue
91                 }

```

```
92             if '0' <= c && c <= '9' {
93                 continue
94             }
95         }
96         b.WriteString(s[written:i])
97         fmt.Fprintf(&b, "%%02x", c)
98         written = i + 1
99     }
100     if written == 0 {
101         return s
102     }
103     b.WriteString(s[written:])
104     return b.String()
105 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/image/format.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package image
6
7 import (
8     "bufio"
9     "errors"
10    "io"
11 )
12
13 // ErrFormat indicates that decoding encountered an unknown
14 var ErrFormat = errors.New("image: unknown format")
15
16 // A format holds an image format's name, magic header and h
17 type format struct {
18     name, magic string
19     decode      func(io.Reader) (Image, error)
20     decodeConfig func(io.Reader) (Config, error)
21 }
22
23 // Formats is the list of registered formats.
24 var formats []format
25
26 // RegisterFormat registers an image format for use by Decod
27 // Name is the name of the format, like "jpeg" or "png".
28 // Magic is the magic prefix that identifies the format's en
29 // string can contain "?" wildcards that each match any one
30 // Decode is the function that decodes the encoded image.
31 // DecodeConfig is the function that decodes just its config
32 func RegisterFormat(name, magic string, decode func(io.Reade
33     formats = append(formats, format{name, magic, decode
34 }
35
36 // A reader is an io.Reader that can also peek ahead.
37 type reader interface {
38     io.Reader
39     Peek(int) ([]byte, error)
40 }
41
42 // AsReader converts an io.Reader to a reader.
43 func asReader(r io.Reader) reader {
44     if rr, ok := r.(reader); ok {
```

```

45         return rr
46     }
47     return bufio.NewReader(r)
48 }
49
50 // Match returns whether magic matches b. Magic may contain
51 func match(magic string, b []byte) bool {
52     if len(magic) != len(b) {
53         return false
54     }
55     for i, c := range b {
56         if magic[i] != c && magic[i] != '?' {
57             return false
58         }
59     }
60     return true
61 }
62
63 // Sniff determines the format of r's data.
64 func sniff(r reader) format {
65     for _, f := range formats {
66         b, err := r.Peek(len(f.magic))
67         if err == nil && match(f.magic, b) {
68             return f
69         }
70     }
71     return format{}
72 }
73
74 // Decode decodes an image that has been encoded in a regist
75 // The string returned is the format name used during format
76 // Format registration is typically done by the init method
77 // specific package.
78 func Decode(r io.Reader) (Image, string, error) {
79     rr := asReader(r)
80     f := sniff(rr)
81     if f.decode == nil {
82         return nil, "", ErrFormat
83     }
84     m, err := f.decode(rr)
85     return m, f.name, err
86 }
87
88 // DecodeConfig decodes the color model and dimensions of an
89 // been encoded in a registered format. The string returned
90 // used during format registration. Format registration is t
91 // the init method of the codec-specific package.
92 func DecodeConfig(r io.Reader) (Config, string, error) {
93     rr := asReader(r)
94     f := sniff(rr)

```

```
95         if f.decodeConfig == nil {
96             return Config{}, "", ErrFormat
97         }
98         c, err := f.decodeConfig(rr)
99         return c, f.name, err
100     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/image/geom.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package image
6
7 import (
8     "strconv"
9 )
10
11 // A Point is an X, Y coordinate pair. The axes increase rig
12 type Point struct {
13     X, Y int
14 }
15
16 // String returns a string representation of p like "(3,4)".
17 func (p Point) String() string {
18     return "(" + strconv.Itoa(p.X) + "," + strconv.Itoa(
19 }
20
21 // Add returns the vector p+q.
22 func (p Point) Add(q Point) Point {
23     return Point{p.X + q.X, p.Y + q.Y}
24 }
25
26 // Sub returns the vector p-q.
27 func (p Point) Sub(q Point) Point {
28     return Point{p.X - q.X, p.Y - q.Y}
29 }
30
31 // Mul returns the vector p*k.
32 func (p Point) Mul(k int) Point {
33     return Point{p.X * k, p.Y * k}
34 }
35
36 // Div returns the vector p/k.
37 func (p Point) Div(k int) Point {
38     return Point{p.X / k, p.Y / k}
39 }
40
41 // In returns whether p is in r.
42 func (p Point) In(r Rectangle) bool {
43     return r.Min.X <= p.X && p.X < r.Max.X &&
44         r.Min.Y <= p.Y && p.Y < r.Max.Y
```

```

45 }
46
47 // Mod returns the point q in r such that p.X-q.X is a multi
48 // and p.Y-q.Y is a multiple of r's height.
49 func (p Point) Mod(r Rectangle) Point {
50     w, h := r.Dx(), r.Dy()
51     p = p.Sub(r.Min)
52     p.X = p.X % w
53     if p.X < 0 {
54         p.X += w
55     }
56     p.Y = p.Y % h
57     if p.Y < 0 {
58         p.Y += h
59     }
60     return p.Add(r.Min)
61 }
62
63 // Eq returns whether p and q are equal.
64 func (p Point) Eq(q Point) bool {
65     return p.X == q.X && p.Y == q.Y
66 }
67
68 // ZP is the zero Point.
69 var ZP Point
70
71 // Pt is shorthand for Point{X, Y}.
72 func Pt(X, Y int) Point {
73     return Point{X, Y}
74 }
75
76 // A Rectangle contains the points with Min.X <= X < Max.X,
77 // It is well-formed if Min.X <= Max.X and likewise for Y. P
78 // well-formed. A rectangle's methods always return well-for
79 // well-formed inputs.
80 type Rectangle struct {
81     Min, Max Point
82 }
83
84 // String returns a string representation of r like "(3,4)-(
85 func (r Rectangle) String() string {
86     return r.Min.String() + "-" + r.Max.String()
87 }
88
89 // Dx returns r's width.
90 func (r Rectangle) Dx() int {
91     return r.Max.X - r.Min.X
92 }
93
94 // Dy returns r's height.

```

```

95 func (r Rectangle) Dy() int {
96     return r.Max.Y - r.Min.Y
97 }
98
99 // Size returns r's width and height.
100 func (r Rectangle) Size() Point {
101     return Point{
102         r.Max.X - r.Min.X,
103         r.Max.Y - r.Min.Y,
104     }
105 }
106
107 // Add returns the rectangle r translated by p.
108 func (r Rectangle) Add(p Point) Rectangle {
109     return Rectangle{
110         Point{r.Min.X + p.X, r.Min.Y + p.Y},
111         Point{r.Max.X + p.X, r.Max.Y + p.Y},
112     }
113 }
114
115 // Sub returns the rectangle r translated by -p.
116 func (r Rectangle) Sub(p Point) Rectangle {
117     return Rectangle{
118         Point{r.Min.X - p.X, r.Min.Y - p.Y},
119         Point{r.Max.X - p.X, r.Max.Y - p.Y},
120     }
121 }
122
123 // Inset returns the rectangle r inset by n, which may be ne
124 // of r's dimensions is less than 2*n then an empty rectangl
125 // of r will be returned.
126 func (r Rectangle) Inset(n int) Rectangle {
127     if r.Dx() < 2*n {
128         r.Min.X = (r.Min.X + r.Max.X) / 2
129         r.Max.X = r.Min.X
130     } else {
131         r.Min.X += n
132         r.Max.X -= n
133     }
134     if r.Dy() < 2*n {
135         r.Min.Y = (r.Min.Y + r.Max.Y) / 2
136         r.Max.Y = r.Min.Y
137     } else {
138         r.Min.Y += n
139         r.Max.Y -= n
140     }
141     return r
142 }
143

```

```

144 // Intersect returns the largest rectangle contained by both
145 // two rectangles do not overlap then the zero rectangle wil
146 func (r Rectangle) Intersect(s Rectangle) Rectangle {
147     if r.Min.X < s.Min.X {
148         r.Min.X = s.Min.X
149     }
150     if r.Min.Y < s.Min.Y {
151         r.Min.Y = s.Min.Y
152     }
153     if r.Max.X > s.Max.X {
154         r.Max.X = s.Max.X
155     }
156     if r.Max.Y > s.Max.Y {
157         r.Max.Y = s.Max.Y
158     }
159     if r.Min.X > r.Max.X || r.Min.Y > r.Max.Y {
160         return ZR
161     }
162     return r
163 }
164
165 // Union returns the smallest rectangle that contains both r
166 func (r Rectangle) Union(s Rectangle) Rectangle {
167     if r.Min.X > s.Min.X {
168         r.Min.X = s.Min.X
169     }
170     if r.Min.Y > s.Min.Y {
171         r.Min.Y = s.Min.Y
172     }
173     if r.Max.X < s.Max.X {
174         r.Max.X = s.Max.X
175     }
176     if r.Max.Y < s.Max.Y {
177         r.Max.Y = s.Max.Y
178     }
179     return r
180 }
181
182 // Empty returns whether the rectangle contains no points.
183 func (r Rectangle) Empty() bool {
184     return r.Min.X >= r.Max.X || r.Min.Y >= r.Max.Y
185 }
186
187 // Eq returns whether r and s are equal.
188 func (r Rectangle) Eq(s Rectangle) bool {
189     return r.Min.X == s.Min.X && r.Min.Y == s.Min.Y &&
190         r.Max.X == s.Max.X && r.Max.Y == s.Max.Y
191 }
192

```

```

193 // Overlaps returns whether r and s have a non-empty interse
194 func (r Rectangle) Overlaps(s Rectangle) bool {
195     return r.Min.X < s.Max.X && s.Min.X < r.Max.X &&
196         r.Min.Y < s.Max.Y && s.Min.Y < r.Max.Y
197 }
198
199 // In returns whether every point in r is in s.
200 func (r Rectangle) In(s Rectangle) bool {
201     if r.Empty() {
202         return true
203     }
204     // Note that r.Max is an exclusive bound for r, so t
205     // does not require that r.Max.In(s).
206     return s.Min.X <= r.Min.X && r.Max.X <= s.Max.X &&
207         s.Min.Y <= r.Min.Y && r.Max.Y <= s.Max.Y
208 }
209
210 // Canon returns the canonical version of r. The returned re
211 // and maximum coordinates swapped if necessary so that it i
212 func (r Rectangle) Canon() Rectangle {
213     if r.Max.X < r.Min.X {
214         r.Min.X, r.Max.X = r.Max.X, r.Min.X
215     }
216     if r.Max.Y < r.Min.Y {
217         r.Min.Y, r.Max.Y = r.Max.Y, r.Min.Y
218     }
219     return r
220 }
221
222 // ZR is the zero Rectangle.
223 var ZR Rectangle
224
225 // Rect is shorthand for Rectangle{Pt(x0, y0), Pt(x1, y1)}.
226 func Rect(x0, y0, x1, y1 int) Rectangle {
227     if x0 > x1 {
228         x0, x1 = x1, x0
229     }
230     if y0 > y1 {
231         y0, y1 = y1, y0
232     }
233     return Rectangle{Point{x0, y0}, Point{x1, y1}}
234 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/image/image.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package image implements a basic 2-D image library.
6 //
7 // The fundamental interface is called Image. An Image conta
8 // are described in the image/color package.
9 //
10 // Values of the Image interface are created either by calli
11 // as NewRGBA and NewPaletted, or by calling Decode on an ic
12 // image data in a format such as GIF, JPEG or PNG. Decoding
13 // image format requires the prior registration of a decoder
14 // Registration is typically automatic as a side effect of i
15 // format's package so that, to decode a PNG image, it suffi
16 //     import _ "image/png"
17 // in a program's main package. The _ means to import a pack
18 // initialization side effects.
19 //
20 // See "The Go image package" for more details:
21 // http://golang.org/doc/articles/image\_package.html
22 package image
23
24 import (
25     "image/color"
26 )
27
28 // Config holds an image's color model and dimensions.
29 type Config struct {
30     ColorModel color.Model
31     Width, Height int
32 }
33
34 // Image is a finite rectangular grid of color.Color values
35 // model.
36 type Image interface {
37     // ColorModel returns the Image's color model.
38     ColorModel() color.Model
39     // Bounds returns the domain for which At can return
40     // The bounds do not necessarily contain the point (
41     Bounds() Rectangle
42     // At returns the color of the pixel at (x, y).
43     // At(Bounds().Min.X, Bounds().Min.Y) returns the up
44     // At(Bounds().Max.X-1, Bounds().Max.Y-1) returns th
```

```

45         At(x, y int) color.Color
46     }
47
48     // PalettedImage is an image whose colors may come from a li
49     // If m is a PalettedImage and m.ColorModel() returns a Pale
50     // then m.At(x, y) should be equivalent to p[m.ColorIndexAt(
51     // color model is not a PalettedColorModel, then ColorIndexA
52     // undefined.
53     type PalettedImage interface {
54         // ColorIndexAt returns the palette index of the pix
55         ColorIndexAt(x, y int) uint8
56         Image
57     }
58
59     // RGBA is an in-memory image whose At method returns color.
60     type RGBA struct {
61         // Pix holds the image's pixels, in R, G, B, A order
62         // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-R
63         Pix []uint8
64         // Stride is the Pix stride (in bytes) between verti
65         Stride int
66         // Rect is the image's bounds.
67         Rect Rectangle
68     }
69
70     func (p *RGBA) ColorModel() color.Model { return color.RGBAM
71
72     func (p *RGBA) Bounds() Rectangle { return p.Rect }
73
74     func (p *RGBA) At(x, y int) color.Color {
75         if !(Point{x, y}.In(p.Rect)) {
76             return color.RGBA{}
77         }
78         i := p.PixOffset(x, y)
79         return color.RGBA{p.Pix[i+0], p.Pix[i+1], p.Pix[i+2]
80     }
81
82     // PixOffset returns the index of the first element of Pix t
83     // the pixel at (x, y).
84     func (p *RGBA) PixOffset(x, y int) int {
85         return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*
86     }
87
88     func (p *RGBA) Set(x, y int, c color.Color) {
89         if !(Point{x, y}.In(p.Rect)) {
90             return
91         }
92         i := p.PixOffset(x, y)
93         c1 := color.RGBAModel.Convert(c).(color.RGBA)
94         p.Pix[i+0] = c1.R

```

```

95         p.Pix[i+1] = c1.G
96         p.Pix[i+2] = c1.B
97         p.Pix[i+3] = c1.A
98     }
99
100 func (p *RGBA) SetRGBA(x, y int, c color.RGBA) {
101     if !(Point{x, y}.In(p.Rect)) {
102         return
103     }
104     i := p.PixOffset(x, y)
105     p.Pix[i+0] = c.R
106     p.Pix[i+1] = c.G
107     p.Pix[i+2] = c.B
108     p.Pix[i+3] = c.A
109 }
110
111 // SubImage returns an image representing the portion of the
112 // through r. The returned value shares pixels with the orig
113 func (p *RGBA) SubImage(r Rectangle) Image {
114     r = r.Intersect(p.Rect)
115     // If r1 and r2 are Rectangles, r1.Intersect(r2) is
116     // either r1 or r2 if the intersection is empty. Wit
117     // this, the Pix[i:] expression below can panic.
118     if r.Empty() {
119         return &RGBA{}
120     }
121     i := p.PixOffset(r.Min.X, r.Min.Y)
122     return &RGBA{
123         Pix:    p.Pix[i:],
124         Stride: p.Stride,
125         Rect:   r,
126     }
127 }
128
129 // Opaque scans the entire image and returns whether or not
130 func (p *RGBA) Opaque() bool {
131     if p.Rect.Empty() {
132         return true
133     }
134     i0, i1 := 3, p.Rect.Dx()*4
135     for y := p.Rect.Min.Y; y < p.Rect.Max.Y; y++ {
136         for i := i0; i < i1; i += 4 {
137             if p.Pix[i] != 0xff {
138                 return false
139             }
140         }
141         i0 += p.Stride
142         i1 += p.Stride
143     }

```

```

144         return true
145     }
146
147     // NewRGBA returns a new RGBA with the given bounds.
148     func NewRGBA(r Rectangle) *RGBA {
149         w, h := r.Dx(), r.Dy()
150         buf := make([]uint8, 4*w*h)
151         return &RGBA{buf, 4 * w, r}
152     }
153
154     // RGBA64 is an in-memory image whose At method returns color
155     type RGBA64 struct {
156         // Pix holds the image's pixels, in R, G, B, A order
157         // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-R
158         Pix []uint8
159         // Stride is the Pix stride (in bytes) between verti
160         Stride int
161         // Rect is the image's bounds.
162         Rect Rectangle
163     }
164
165     func (p *RGBA64) ColorModel() color.Model { return color.RGB
166
167     func (p *RGBA64) Bounds() Rectangle { return p.Rect }
168
169     func (p *RGBA64) At(x, y int) color.Color {
170         if !(Point{x, y}.In(p.Rect)) {
171             return color.RGBA64{}
172         }
173         i := p.PixOffset(x, y)
174         return color.RGBA64{
175             uint16(p.Pix[i+0])<<8 | uint16(p.Pix[i+1]),
176             uint16(p.Pix[i+2])<<8 | uint16(p.Pix[i+3]),
177             uint16(p.Pix[i+4])<<8 | uint16(p.Pix[i+5]),
178             uint16(p.Pix[i+6])<<8 | uint16(p.Pix[i+7]),
179         }
180     }
181
182     // PixOffset returns the index of the first element of Pix t
183     // the pixel at (x, y).
184     func (p *RGBA64) PixOffset(x, y int) int {
185         return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*
186     }
187
188     func (p *RGBA64) Set(x, y int, c color.Color) {
189         if !(Point{x, y}.In(p.Rect)) {
190             return
191         }
192         i := p.PixOffset(x, y)

```

```

193         c1 := color.RGBA64Model.Convert(c).(color.RGBA64)
194         p.Pix[i+0] = uint8(c1.R >> 8)
195         p.Pix[i+1] = uint8(c1.R)
196         p.Pix[i+2] = uint8(c1.G >> 8)
197         p.Pix[i+3] = uint8(c1.G)
198         p.Pix[i+4] = uint8(c1.B >> 8)
199         p.Pix[i+5] = uint8(c1.B)
200         p.Pix[i+6] = uint8(c1.A >> 8)
201         p.Pix[i+7] = uint8(c1.A)
202     }
203
204     func (p *RGBA64) SetRGBA64(x, y int, c color.RGBA64) {
205         if !(Point{x, y}.In(p.Rect)) {
206             return
207         }
208         i := p.PixOffset(x, y)
209         p.Pix[i+0] = uint8(c.R >> 8)
210         p.Pix[i+1] = uint8(c.R)
211         p.Pix[i+2] = uint8(c.G >> 8)
212         p.Pix[i+3] = uint8(c.G)
213         p.Pix[i+4] = uint8(c.B >> 8)
214         p.Pix[i+5] = uint8(c.B)
215         p.Pix[i+6] = uint8(c.A >> 8)
216         p.Pix[i+7] = uint8(c.A)
217     }
218
219     // SubImage returns an image representing the portion of the
220     // through r. The returned value shares pixels with the orig
221     func (p *RGBA64) SubImage(r Rectangle) Image {
222         r = r.Intersect(p.Rect)
223         // If r1 and r2 are Rectangles, r1.Intersect(r2) is
224         // either r1 or r2 if the intersection is empty. Wit
225         // this, the Pix[i:] expression below can panic.
226         if r.Empty() {
227             return &RGBA64{}
228         }
229         i := p.PixOffset(r.Min.X, r.Min.Y)
230         return &RGBA64{
231             Pix:    p.Pix[i:],
232             Stride: p.Stride,
233             Rect:    r,
234         }
235     }
236
237     // Opaque scans the entire image and returns whether or not
238     func (p *RGBA64) Opaque() bool {
239         if p.Rect.Empty() {
240             return true
241         }
242         i0, i1 := 6, p.Rect.Dx()*8

```

```

243     for y := p.Rect.Min.Y; y < p.Rect.Max.Y; y++ {
244         for i := i0; i < i1; i += 8 {
245             if p.Pix[i+0] != 0xff || p.Pix[i+1]
246                 return false
247         }
248     }
249     i0 += p.Stride
250     i1 += p.Stride
251 }
252 return true
253 }
254
255 // NewRGBA64 returns a new RGBA64 with the given bounds.
256 func NewRGBA64(r Rectangle) *RGBA64 {
257     w, h := r.Dx(), r.Dy()
258     pix := make([]uint8, 8*w*h)
259     return &RGBA64{pix, 8 * w, r}
260 }
261
262 // NRGBA is an in-memory image whose At method returns color
263 type NRGBA struct {
264     // Pix holds the image's pixels, in R, G, B, A order
265     // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-R
266     Pix []uint8
267     // Stride is the Pix stride (in bytes) between verti
268     Stride int
269     // Rect is the image's bounds.
270     Rect Rectangle
271 }
272
273 func (p *NRGBA) ColorModel() color.Model { return color.NRGE
274
275 func (p *NRGBA) Bounds() Rectangle { return p.Rect }
276
277 func (p *NRGBA) At(x, y int) color.Color {
278     if !(Point{x, y}.In(p.Rect)) {
279         return color.NRGBA{}
280     }
281     i := p.PixOffset(x, y)
282     return color.NRGBA{p.Pix[i+0], p.Pix[i+1], p.Pix[i+2]
283 }
284
285 // PixOffset returns the index of the first element of Pix t
286 // the pixel at (x, y).
287 func (p *NRGBA) PixOffset(x, y int) int {
288     return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*
289 }
290
291 func (p *NRGBA) Set(x, y int, c color.Color) {

```

```

292         if !(Point{x, y}.In(p.Rect)) {
293             return
294         }
295         i := p.PixOffset(x, y)
296         c1 := color.NRGBAModel.Convert(c).(color.NRGBA)
297         p.Pix[i+0] = c1.R
298         p.Pix[i+1] = c1.G
299         p.Pix[i+2] = c1.B
300         p.Pix[i+3] = c1.A
301     }
302
303     func (p *NRGBA) SetNRGBA(x, y int, c color.NRGBA) {
304         if !(Point{x, y}.In(p.Rect)) {
305             return
306         }
307         i := p.PixOffset(x, y)
308         p.Pix[i+0] = c.R
309         p.Pix[i+1] = c.G
310         p.Pix[i+2] = c.B
311         p.Pix[i+3] = c.A
312     }
313
314     // SubImage returns an image representing the portion of the
315     // through r. The returned value shares pixels with the orig
316     func (p *NRGBA) SubImage(r Rectangle) Image {
317         r = r.Intersect(p.Rect)
318         // If r1 and r2 are Rectangles, r1.Intersect(r2) is
319         // either r1 or r2 if the intersection is empty. Wit
320         // this, the Pix[i:] expression below can panic.
321         if r.Empty() {
322             return &NRGBA{}
323         }
324         i := p.PixOffset(r.Min.X, r.Min.Y)
325         return &NRGBA{
326             Pix:    p.Pix[i:],
327             Stride: p.Stride,
328             Rect:   r,
329         }
330     }
331
332     // Opaque scans the entire image and returns whether or not
333     func (p *NRGBA) Opaque() bool {
334         if p.Rect.Empty() {
335             return true
336         }
337         i0, i1 := 3, p.Rect.Dx()*4
338         for y := p.Rect.Min.Y; y < p.Rect.Max.Y; y++ {
339             for i := i0; i < i1; i += 4 {
340                 if p.Pix[i] != 0xff {

```

```

341                                     return false
342                                     }
343                                 }
344                                 i0 += p.Stride
345                                 i1 += p.Stride
346                             }
347                             return true
348     }
349
350 // NewNRGBA returns a new NRGBA with the given bounds.
351 func NewNRGBA(r Rectangle) *NRGBA {
352     w, h := r.Dx(), r.Dy()
353     pix := make([]uint8, 4*w*h)
354     return &NRGBA{pix, 4 * w, r}
355 }
356
357 // NRGBA64 is an in-memory image whose At method returns col
358 type NRGBA64 struct {
359     // Pix holds the image's pixels, in R, G, B, A order
360     // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-R
361     Pix []uint8
362     // Stride is the Pix stride (in bytes) between verti
363     Stride int
364     // Rect is the image's bounds.
365     Rect Rectangle
366 }
367
368 func (p *NRGBA64) ColorModel() color.Model { return color.NR
369
370 func (p *NRGBA64) Bounds() Rectangle { return p.Rect }
371
372 func (p *NRGBA64) At(x, y int) color.Color {
373     if !(Point{x, y}.In(p.Rect)) {
374         return color.NRGBA64{}
375     }
376     i := p.PixOffset(x, y)
377     return color.NRGBA64{
378         uint16(p.Pix[i+0])<<8 | uint16(p.Pix[i+1]),
379         uint16(p.Pix[i+2])<<8 | uint16(p.Pix[i+3]),
380         uint16(p.Pix[i+4])<<8 | uint16(p.Pix[i+5]),
381         uint16(p.Pix[i+6])<<8 | uint16(p.Pix[i+7]),
382     }
383 }
384
385 // PixOffset returns the index of the first element of Pix t
386 // the pixel at (x, y).
387 func (p *NRGBA64) PixOffset(x, y int) int {
388     return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*
389 }
390

```

```

391 func (p *NRGBA64) Set(x, y int, c color.Color) {
392     if !(Point{x, y}.In(p.Rect)) {
393         return
394     }
395     i := p.PixOffset(x, y)
396     c1 := color.NRGBA64Model.Convert(c).(color.NRGBA64)
397     p.Pix[i+0] = uint8(c1.R >> 8)
398     p.Pix[i+1] = uint8(c1.R)
399     p.Pix[i+2] = uint8(c1.G >> 8)
400     p.Pix[i+3] = uint8(c1.G)
401     p.Pix[i+4] = uint8(c1.B >> 8)
402     p.Pix[i+5] = uint8(c1.B)
403     p.Pix[i+6] = uint8(c1.A >> 8)
404     p.Pix[i+7] = uint8(c1.A)
405 }
406
407 func (p *NRGBA64) SetNRGBA64(x, y int, c color.NRGBA64) {
408     if !(Point{x, y}.In(p.Rect)) {
409         return
410     }
411     i := p.PixOffset(x, y)
412     p.Pix[i+0] = uint8(c.R >> 8)
413     p.Pix[i+1] = uint8(c.R)
414     p.Pix[i+2] = uint8(c.G >> 8)
415     p.Pix[i+3] = uint8(c.G)
416     p.Pix[i+4] = uint8(c.B >> 8)
417     p.Pix[i+5] = uint8(c.B)
418     p.Pix[i+6] = uint8(c.A >> 8)
419     p.Pix[i+7] = uint8(c.A)
420 }
421
422 // SubImage returns an image representing the portion of the
423 // through r. The returned value shares pixels with the orig
424 func (p *NRGBA64) SubImage(r Rectangle) Image {
425     r = r.Intersect(p.Rect)
426     // If r1 and r2 are Rectangles, r1.Intersect(r2) is
427     // either r1 or r2 if the intersection is empty. Wit
428     // this, the Pix[i:] expression below can panic.
429     if r.Empty() {
430         return &NRGBA64{}
431     }
432     i := p.PixOffset(r.Min.X, r.Min.Y)
433     return &NRGBA64{
434         Pix:    p.Pix[i:],
435         Stride: p.Stride,
436         Rect:   r,
437     }
438 }
439

```

```

440 // Opaque scans the entire image and returns whether or not
441 func (p *NRGBA64) Opaque() bool {
442     if p.Rect.Empty() {
443         return true
444     }
445     i0, i1 := 6, p.Rect.Dx()*8
446     for y := p.Rect.Min.Y; y < p.Rect.Max.Y; y++ {
447         for i := i0; i < i1; i += 8 {
448             if p.Pix[i+0] != 0xff || p.Pix[i+1]
449                 return false
450         }
451     }
452     i0 += p.Stride
453     i1 += p.Stride
454 }
455 return true
456 }
457
458 // NewNRGBA64 returns a new NRGBA64 with the given bounds.
459 func NewNRGBA64(r Rectangle) *NRGBA64 {
460     w, h := r.Dx(), r.Dy()
461     pix := make([]uint8, 8*w*h)
462     return &NRGBA64{pix, 8 * w, r}
463 }
464
465 // Alpha is an in-memory image whose At method returns color
466 type Alpha struct {
467     // Pix holds the image's pixels, as alpha values. Th
468     // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-R
469     Pix []uint8
470     // Stride is the Pix stride (in bytes) between verti
471     Stride int
472     // Rect is the image's bounds.
473     Rect Rectangle
474 }
475
476 func (p *Alpha) ColorModel() color.Model { return color.Alph
477
478 func (p *Alpha) Bounds() Rectangle { return p.Rect }
479
480 func (p *Alpha) At(x, y int) color.Color {
481     if !(Point{x, y}.In(p.Rect)) {
482         return color.Alpha{}
483     }
484     i := p.PixOffset(x, y)
485     return color.Alpha{p.Pix[i]}
486 }
487
488 // PixOffset returns the index of the first element of Pix t

```

```

489 // the pixel at (x, y).
490 func (p *Alpha) PixOffset(x, y int) int {
491     return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*
492 }
493
494 func (p *Alpha) Set(x, y int, c color.Color) {
495     if !(Point{x, y}.In(p.Rect)) {
496         return
497     }
498     i := p.PixOffset(x, y)
499     p.Pix[i] = color.AlphaModel.Convert(c).(color.Alpha)
500 }
501
502 func (p *Alpha) SetAlpha(x, y int, c color.Alpha) {
503     if !(Point{x, y}.In(p.Rect)) {
504         return
505     }
506     i := p.PixOffset(x, y)
507     p.Pix[i] = c.A
508 }
509
510 // SubImage returns an image representing the portion of the
511 // through r. The returned value shares pixels with the orig
512 func (p *Alpha) SubImage(r Rectangle) Image {
513     r = r.Intersect(p.Rect)
514     // If r1 and r2 are Rectangles, r1.Intersect(r2) is
515     // either r1 or r2 if the intersection is empty. Wit
516     // this, the Pix[i:] expression below can panic.
517     if r.Empty() {
518         return &Alpha{}
519     }
520     i := p.PixOffset(r.Min.X, r.Min.Y)
521     return &Alpha{
522         Pix:    p.Pix[i:],
523         Stride: p.Stride,
524         Rect:   r,
525     }
526 }
527
528 // Opaque scans the entire image and returns whether or not
529 func (p *Alpha) Opaque() bool {
530     if p.Rect.Empty() {
531         return true
532     }
533     i0, i1 := 0, p.Rect.Dx()
534     for y := p.Rect.Min.Y; y < p.Rect.Max.Y; y++ {
535         for i := i0; i < i1; i++ {
536             if p.Pix[i] != 0xff {
537                 return false
538             }

```

```

539         }
540         i0 += p.Stride
541         i1 += p.Stride
542     }
543     return true
544 }
545
546 // NewAlpha returns a new Alpha with the given bounds.
547 func NewAlpha(r Rectangle) *Alpha {
548     w, h := r.Dx(), r.Dy()
549     pix := make([]uint8, 1*w*h)
550     return &Alpha{pix, 1 * w, r}
551 }
552
553 // Alpha16 is an in-memory image whose At method returns col
554 type Alpha16 struct {
555     // Pix holds the image's pixels, as alpha values in
556     // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-R
557     Pix []uint8
558     // Stride is the Pix stride (in bytes) between verti
559     Stride int
560     // Rect is the image's bounds.
561     Rect Rectangle
562 }
563
564 func (p *Alpha16) ColorModel() color.Model { return color.AL
565
566 func (p *Alpha16) Bounds() Rectangle { return p.Rect }
567
568 func (p *Alpha16) At(x, y int) color.Color {
569     if !(Point{x, y}.In(p.Rect)) {
570         return color.Alpha16{}
571     }
572     i := p.PixOffset(x, y)
573     return color.Alpha16{uint16(p.Pix[i+0])<<8 | uint16(
574 }
575
576 // PixOffset returns the index of the first element of Pix t
577 // the pixel at (x, y).
578 func (p *Alpha16) PixOffset(x, y int) int {
579     return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*
580 }
581
582 func (p *Alpha16) Set(x, y int, c color.Color) {
583     if !(Point{x, y}.In(p.Rect)) {
584         return
585     }
586     i := p.PixOffset(x, y)
587     c1 := color.Alpha16Model.Convert(c).(color.Alpha16)

```

```

588         p.Pix[i+0] = uint8(c1.A >> 8)
589         p.Pix[i+1] = uint8(c1.A)
590     }
591
592     func (p *Alpha16) SetAlpha16(x, y int, c color.Alpha16) {
593         if !(Point{x, y}.In(p.Rect)) {
594             return
595         }
596         i := p.PixOffset(x, y)
597         p.Pix[i+0] = uint8(c.A >> 8)
598         p.Pix[i+1] = uint8(c.A)
599     }
600
601     // SubImage returns an image representing the portion of the
602     // through r. The returned value shares pixels with the orig
603     func (p *Alpha16) SubImage(r Rectangle) Image {
604         r = r.Intersect(p.Rect)
605         // If r1 and r2 are Rectangles, r1.Intersect(r2) is
606         // either r1 or r2 if the intersection is empty. Wit
607         // this, the Pix[i:] expression below can panic.
608         if r.Empty() {
609             return &Alpha16{}
610         }
611         i := p.PixOffset(r.Min.X, r.Min.Y)
612         return &Alpha16{
613             Pix:    p.Pix[i:],
614             Stride: p.Stride,
615             Rect:   r,
616         }
617     }
618
619     // Opaque scans the entire image and returns whether or not
620     func (p *Alpha16) Opaque() bool {
621         if p.Rect.Empty() {
622             return true
623         }
624         i0, i1 := 0, p.Rect.Dx()*2
625         for y := p.Rect.Min.Y; y < p.Rect.Max.Y; y++ {
626             for i := i0; i < i1; i += 2 {
627                 if p.Pix[i+0] != 0xff || p.Pix[i+1]
628                     return false
629             }
630         }
631         i0 += p.Stride
632         i1 += p.Stride
633     }
634     return true
635 }
636

```

```

637 // NewAlpha16 returns a new Alpha16 with the given bounds.
638 func NewAlpha16(r Rectangle) *Alpha16 {
639     w, h := r.Dx(), r.Dy()
640     pix := make([]uint8, 2*w*h)
641     return &Alpha16{pix, 2 * w, r}
642 }
643
644 // Gray is an in-memory image whose At method returns color.
645 type Gray struct {
646     // Pix holds the image's pixels, as gray values. The
647     // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-R
648     Pix []uint8
649     // Stride is the Pix stride (in bytes) between verti
650     Stride int
651     // Rect is the image's bounds.
652     Rect Rectangle
653 }
654
655 func (p *Gray) ColorModel() color.Model { return color.GrayM
656
657 func (p *Gray) Bounds() Rectangle { return p.Rect }
658
659 func (p *Gray) At(x, y int) color.Color {
660     if !(Point{x, y}.In(p.Rect)) {
661         return color.Gray{}
662     }
663     i := p.PixOffset(x, y)
664     return color.Gray{p.Pix[i]}
665 }
666
667 // PixOffset returns the index of the first element of Pix t
668 // the pixel at (x, y).
669 func (p *Gray) PixOffset(x, y int) int {
670     return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*
671 }
672
673 func (p *Gray) Set(x, y int, c color.Color) {
674     if !(Point{x, y}.In(p.Rect)) {
675         return
676     }
677     i := p.PixOffset(x, y)
678     p.Pix[i] = color.GrayModel.Convert(c).(color.Gray).Y
679 }
680
681 func (p *Gray) SetGray(x, y int, c color.Gray) {
682     if !(Point{x, y}.In(p.Rect)) {
683         return
684     }
685     i := p.PixOffset(x, y)
686     p.Pix[i] = c.Y

```

```

687 }
688
689 // SubImage returns an image representing the portion of the
690 // through r. The returned value shares pixels with the orig
691 func (p *Gray) SubImage(r Rectangle) Image {
692     r = r.Intersect(p.Rect)
693     // If r1 and r2 are Rectangles, r1.Intersect(r2) is
694     // either r1 or r2 if the intersection is empty. Wit
695     // this, the Pix[i:] expression below can panic.
696     if r.Empty() {
697         return &Gray{}
698     }
699     i := p.PixOffset(r.Min.X, r.Min.Y)
700     return &Gray{
701         Pix:    p.Pix[i:],
702         Stride: p.Stride,
703         Rect:   r,
704     }
705 }
706
707 // Opaque scans the entire image and returns whether or not
708 func (p *Gray) Opaque() bool {
709     return true
710 }
711
712 // NewGray returns a new Gray with the given bounds.
713 func NewGray(r Rectangle) *Gray {
714     w, h := r.Dx(), r.Dy()
715     pix := make([]uint8, 1*w*h)
716     return &Gray{pix, 1 * w, r}
717 }
718
719 // Gray16 is an in-memory image whose At method returns color
720 type Gray16 struct {
721     // Pix holds the image's pixels, as gray values in b
722     // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-R
723     Pix []uint8
724     // Stride is the Pix stride (in bytes) between verti
725     Stride int
726     // Rect is the image's bounds.
727     Rect Rectangle
728 }
729
730 func (p *Gray16) ColorModel() color.Model { return color.Gra
731
732 func (p *Gray16) Bounds() Rectangle { return p.Rect }
733
734 func (p *Gray16) At(x, y int) color.Color {
735     if !(Point{x, y}.In(p.Rect)) {

```

```

736         return color.Gray16{}
737     }
738     i := p.PixOffset(x, y)
739     return color.Gray16{uint16(p.Pix[i+0])<<8 | uint16(p
740 }
741
742 // PixOffset returns the index of the first element of Pix t
743 // the pixel at (x, y).
744 func (p *Gray16) PixOffset(x, y int) int {
745     return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*
746 }
747
748 func (p *Gray16) Set(x, y int, c color.Color) {
749     if !(Point{x, y}.In(p.Rect)) {
750         return
751     }
752     i := p.PixOffset(x, y)
753     c1 := color.Gray16Model.Convert(c).(color.Gray16)
754     p.Pix[i+0] = uint8(c1.Y >> 8)
755     p.Pix[i+1] = uint8(c1.Y)
756 }
757
758 func (p *Gray16) SetGray16(x, y int, c color.Gray16) {
759     if !(Point{x, y}.In(p.Rect)) {
760         return
761     }
762     i := p.PixOffset(x, y)
763     p.Pix[i+0] = uint8(c.Y >> 8)
764     p.Pix[i+1] = uint8(c.Y)
765 }
766
767 // SubImage returns an image representing the portion of the
768 // through r. The returned value shares pixels with the orig
769 func (p *Gray16) SubImage(r Rectangle) Image {
770     r = r.Intersect(p.Rect)
771     // If r1 and r2 are Rectangles, r1.Intersect(r2) is
772     // either r1 or r2 if the intersection is empty. Wit
773     // this, the Pix[i:] expression below can panic.
774     if r.Empty() {
775         return &Gray16{}
776     }
777     i := p.PixOffset(r.Min.X, r.Min.Y)
778     return &Gray16{
779         Pix:    p.Pix[i:],
780         Stride: p.Stride,
781         Rect:   r,
782     }
783 }
784

```

```

785 // Opaque scans the entire image and returns whether or not
786 func (p *Gray16) Opaque() bool {
787     return true
788 }
789
790 // NewGray16 returns a new Gray16 with the given bounds.
791 func NewGray16(r Rectangle) *Gray16 {
792     w, h := r.Dx(), r.Dy()
793     pix := make([]uint8, 2*w*h)
794     return &Gray16{pix, 2 * w, r}
795 }
796
797 // Paletted is an in-memory image of uint8 indices into a gi
798 type Paletted struct {
799     // Pix holds the image's pixels, as palette indices.
800     // (x, y) starts at Pix[(y-Rect.Min.Y)*Stride + (x-R
801     Pix []uint8
802     // Stride is the Pix stride (in bytes) between verti
803     Stride int
804     // Rect is the image's bounds.
805     Rect Rectangle
806     // Palette is the image's palette.
807     Palette color.Palette
808 }
809
810 func (p *Paletted) ColorModel() color.Model { return p.Palet
811
812 func (p *Paletted) Bounds() Rectangle { return p.Rect }
813
814 func (p *Paletted) At(x, y int) color.Color {
815     if len(p.Palette) == 0 {
816         return nil
817     }
818     if !(Point{x, y}.In(p.Rect)) {
819         return p.Palette[0]
820     }
821     i := p.PixOffset(x, y)
822     return p.Palette[p.Pix[i]]
823 }
824
825 // PixOffset returns the index of the first element of Pix t
826 // the pixel at (x, y).
827 func (p *Paletted) PixOffset(x, y int) int {
828     return (y-p.Rect.Min.Y)*p.Stride + (x-p.Rect.Min.X)*
829 }
830
831 func (p *Paletted) Set(x, y int, c color.Color) {
832     if !(Point{x, y}.In(p.Rect)) {
833         return
834     }

```

```

835         i := p.PixOffset(x, y)
836         p.Pix[i] = uint8(p.Palette.Index(c))
837     }
838
839     func (p *Paletted) ColorIndexAt(x, y int) uint8 {
840         if !(Point{x, y}.In(p.Rect)) {
841             return 0
842         }
843         i := p.PixOffset(x, y)
844         return p.Pix[i]
845     }
846
847     func (p *Paletted) SetColorIndex(x, y int, index uint8) {
848         if !(Point{x, y}.In(p.Rect)) {
849             return
850         }
851         i := p.PixOffset(x, y)
852         p.Pix[i] = index
853     }
854
855     // SubImage returns an image representing the portion of the
856     // through r. The returned value shares pixels with the orig
857     func (p *Paletted) SubImage(r Rectangle) Image {
858         r = r.Intersect(p.Rect)
859         // If r1 and r2 are Rectangles, r1.Intersect(r2) is
860         // either r1 or r2 if the intersection is empty. Wit
861         // this, the Pix[i:] expression below can panic.
862         if r.Empty() {
863             return &Paletted{
864                 Palette: p.Palette,
865             }
866         }
867         i := p.PixOffset(r.Min.X, r.Min.Y)
868         return &Paletted{
869             Pix:      p.Pix[i:],
870             Stride:   p.Stride,
871             Rect:     p.Rect.Intersect(r),
872             Palette:  p.Palette,
873         }
874     }
875
876     // Opaque scans the entire image and returns whether or not
877     func (p *Paletted) Opaque() bool {
878         var present [256]bool
879         i0, i1 := 0, p.Rect.Dx()
880         for y := p.Rect.Min.Y; y < p.Rect.Max.Y; y++ {
881             for _, c := range p.Pix[i0:i1] {
882                 present[c] = true
883             }

```

```

884         i0 += p.Stride
885         i1 += p.Stride
886     }
887     for i, c := range p.Palette {
888         if !present[i] {
889             continue
890         }
891         _, _, _, a := c.RGBA()
892         if a != 0xffff {
893             return false
894         }
895     }
896     return true
897 }
898
899 // NewPaletted returns a new Paletted with the given width,
900 func NewPaletted(r Rectangle, p color.Palette) *Paletted {
901     w, h := r.Dx(), r.Dy()
902     pix := make([]uint8, 1*w*h)
903     return &Paletted{pix, 1 * w, r, p}
904 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/image/names.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package image
6
7 import (
8     "image/color"
9 )
10
11 var (
12     // Black is an opaque black uniform image.
13     Black = NewUniform(color.Black)
14     // White is an opaque white uniform image.
15     White = NewUniform(color.White)
16     // Transparent is a fully transparent uniform image.
17     Transparent = NewUniform(color.Transparent)
18     // Opaque is a fully opaque uniform image.
19     Opaque = NewUniform(color.Opaque)
20 )
21
22 // Uniform is an infinite-sized Image of uniform color.
23 // It implements the color.Color, color.ColorModel, and Imag
24 type Uniform struct {
25     C color.Color
26 }
27
28 func (c *Uniform) RGBA() (r, g, b, a uint32) {
29     return c.C.RGBA()
30 }
31
32 func (c *Uniform) ColorModel() color.Model {
33     return c
34 }
35
36 func (c *Uniform) Convert(color.Color) color.Color {
37     return c.C
38 }
39
40 func (c *Uniform) Bounds() Rectangle { return Rectangle{Poin
41
42 func (c *Uniform) At(x, y int) color.Color { return c.C }
43
44 // Opaque scans the entire image and returns whether or not
```

```
45 func (c *Uniform) Opaque() bool {
46     _/ _/ _/ a := c.C.RGBA()
47     return a == 0xffff
48 }
49
50 func NewUniform(c color.Color) *Uniform {
51     return &Uniform{c}
52 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/image/ycbcr.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package image
6
7 import (
8     "image/color"
9 )
10
11 // YCbCrSubsampleRatio is the chroma subsample ratio used in
12 type YCbCrSubsampleRatio int
13
14 const (
15     YCbCrSubsampleRatio444 YCbCrSubsampleRatio = iota
16     YCbCrSubsampleRatio422
17     YCbCrSubsampleRatio420
18 )
19
20 func (s YCbCrSubsampleRatio) String() string {
21     switch s {
22     case YCbCrSubsampleRatio444:
23         return "YCbCrSubsampleRatio444"
24     case YCbCrSubsampleRatio422:
25         return "YCbCrSubsampleRatio422"
26     case YCbCrSubsampleRatio420:
27         return "YCbCrSubsampleRatio420"
28     }
29     return "YCbCrSubsampleRatioUnknown"
30 }
31
32 // YCbCr is an in-memory image of Y'CbCr colors. There is one
33 // pixel, but each Cb and Cr sample can span one or more pixels.
34 // YStride is the Y slice index delta between vertically adjacent
35 // CStride is the Cb and Cr slice index delta between vertically
36 // adjacent samples.
37 // It is not an absolute requirement, but YStride and len(Y)
38 // must be multiples of 8, and:
39 //     For 4:4:4, CStride == YStride/1 && len(Cb) == len(Cr)
40 //     For 4:2:2, CStride == YStride/2 && len(Cb) == len(Cr)
41 //     For 4:2:0, CStride == YStride/2 && len(Cb) == len(Cr)
42 type YCbCr struct {
43     Y, Cb, Cr []uint8
44     YStride   int
```

```

45         CStride          int
46         SubsampleRatio  YCbCrSubsampleRatio
47         Rect             Rectangle
48     }
49
50     func (p *YCbCr) ColorModel() color.Model {
51         return color.YCbCrModel
52     }
53
54     func (p *YCbCr) Bounds() Rectangle {
55         return p.Rect
56     }
57
58     func (p *YCbCr) At(x, y int) color.Color {
59         if !(Point{x, y}.In(p.Rect)) {
60             return color.YCbCr{}
61         }
62         yi := p.YOffset(x, y)
63         ci := p.COffset(x, y)
64         return color.YCbCr{
65             p.Y[yi],
66             p.Cb[ci],
67             p.Cr[ci],
68         }
69     }
70
71     // YOffset returns the index of the first element of Y that
72     // the pixel at (x, y).
73     func (p *YCbCr) YOffset(x, y int) int {
74         return (y-p.Rect.Min.Y)*p.YStride + (x - p.Rect.Min.
75     }
76
77     // COffset returns the index of the first element of Cb or C
78     // to the pixel at (x, y).
79     func (p *YCbCr) COffset(x, y int) int {
80         switch p.SubsampleRatio {
81             case YCbCrSubsampleRatio422:
82                 return (y-p.Rect.Min.Y)*p.CStride + (x/2 - p
83             case YCbCrSubsampleRatio420:
84                 return (y/2-p.Rect.Min.Y/2)*p.CStride + (x/2
85         }
86         // Default to 4:4:4 subsampling.
87         return (y-p.Rect.Min.Y)*p.CStride + (x - p.Rect.Min.
88     }
89
90     // SubImage returns an image representing the portion of the
91     // through r. The returned value shares pixels with the orig
92     func (p *YCbCr) SubImage(r Rectangle) Image {
93         r = r.Intersect(p.Rect)
94         // If r1 and r2 are Rectangles, r1.Intersect(r2) is

```

```

95         // either r1 or r2 if the intersection is empty. Wit
96         // this, the Pix[i:] expression below can panic.
97         if r.Empty() {
98             return &YCbCr{
99                 SubsampleRatio: p.SubsampleRatio,
100            }
101        }
102        yi := p.YOffset(r.Min.X, r.Min.Y)
103        ci := p.COffset(r.Min.X, r.Min.Y)
104        return &YCbCr{
105            Y:          p.Y[yi:],
106            Cb:         p.Cb[ci:],
107            Cr:         p.Cr[ci:],
108            SubsampleRatio: p.SubsampleRatio,
109            YStride:    p.YStride,
110            CStride:    p.CStride,
111            Rect:       r,
112        }
113    }
114
115    func (p *YCbCr) Opaque() bool {
116        return true
117    }
118
119    // NewYCbCr returns a new YCbCr with the given bounds and su
120    func NewYCbCr(r Rectangle, subsampleRatio YCbCrSubsampleRati
121        w, h, cw, ch := r.Dx(), r.Dy(), 0, 0
122        switch subsampleRatio {
123        case YCbCrSubsampleRatio422:
124            cw = (r.Max.X+1)/2 - r.Min.X/2
125            ch = h
126        case YCbCrSubsampleRatio420:
127            cw = (r.Max.X+1)/2 - r.Min.X/2
128            ch = (r.Max.Y+1)/2 - r.Min.Y/2
129        default:
130            // Default to 4:4:4 subsampling.
131            cw = w
132            ch = h
133        }
134        b := make([]byte, w*h+2*cw*ch)
135        return &YCbCr{
136            Y:          b[:w*h],
137            Cb:         b[w*h+0*cw*ch : w*h+1*cw*ch],
138            Cr:         b[w*h+1*cw*ch : w*h+2*cw*ch],
139            SubsampleRatio: subsampleRatio,
140            YStride:    w,
141            CStride:    cw,
142            Rect:       r,
143        }

```

144 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/image/color/color.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package color implements a basic color library.
6 package color
7
8 // Color can convert itself to alpha-premultiplied 16-bits p
9 // The conversion may be lossy.
10 type Color interface {
11     // RGBA returns the alpha-premultiplied red, green,
12     // for the color. Each value ranges within [0, 0xFF
13     // by a uint32 so that multiplying by a blend factor
14     // overflow.
15     RGBA() (r, g, b, a uint32)
16 }
17
18 // RGBA represents a traditional 32-bit alpha-premultiplied
19 // having 8 bits for each of red, green, blue and alpha.
20 type RGBA struct {
21     R, G, B, A uint8
22 }
23
24 func (c RGBA) RGBA() (r, g, b, a uint32) {
25     r = uint32(c.R)
26     r |= r << 8
27     g = uint32(c.G)
28     g |= g << 8
29     b = uint32(c.B)
30     b |= b << 8
31     a = uint32(c.A)
32     a |= a << 8
33     return
34 }
35
36 // RGBA64 represents a 64-bit alpha-premultiplied color,
37 // having 16 bits for each of red, green, blue and alpha.
38 type RGBA64 struct {
39     R, G, B, A uint16
40 }
41
```

```

42 func (c RGBA64) RGBA() (r, g, b, a uint32) {
43     return uint32(c.R), uint32(c.G), uint32(c.B), uint32
44 }
45
46 // NRGBA represents a non-alpha-premultiplied 32-bit color.
47 type NRGBA struct {
48     R, G, B, A uint8
49 }
50
51 func (c NRGBA) RGBA() (r, g, b, a uint32) {
52     r = uint32(c.R)
53     r |= r << 8
54     r *= uint32(c.A)
55     r /= 0xff
56     g = uint32(c.G)
57     g |= g << 8
58     g *= uint32(c.A)
59     g /= 0xff
60     b = uint32(c.B)
61     b |= b << 8
62     b *= uint32(c.A)
63     b /= 0xff
64     a = uint32(c.A)
65     a |= a << 8
66     return
67 }
68
69 // NRGBA64 represents a non-alpha-premultiplied 64-bit color
70 // having 16 bits for each of red, green, blue and alpha.
71 type NRGBA64 struct {
72     R, G, B, A uint16
73 }
74
75 func (c NRGBA64) RGBA() (r, g, b, a uint32) {
76     r = uint32(c.R)
77     r *= uint32(c.A)
78     r /= 0xffff
79     g = uint32(c.G)
80     g *= uint32(c.A)
81     g /= 0xffff
82     b = uint32(c.B)
83     b *= uint32(c.A)
84     b /= 0xffff
85     a = uint32(c.A)
86     return
87 }
88
89 // Alpha represents an 8-bit alpha color.
90 type Alpha struct {
91     A uint8

```

```

92 }
93
94 func (c Alpha) RGBA() (r, g, b, a uint32) {
95     a = uint32(c.A)
96     a |= a << 8
97     return a, a, a, a
98 }
99
100 // Alpha16 represents a 16-bit alpha color.
101 type Alpha16 struct {
102     A uint16
103 }
104
105 func (c Alpha16) RGBA() (r, g, b, a uint32) {
106     a = uint32(c.A)
107     return a, a, a, a
108 }
109
110 // Gray represents an 8-bit grayscale color.
111 type Gray struct {
112     Y uint8
113 }
114
115 func (c Gray) RGBA() (r, g, b, a uint32) {
116     y := uint32(c.Y)
117     y |= y << 8
118     return y, y, y, 0xffff
119 }
120
121 // Gray16 represents a 16-bit grayscale color.
122 type Gray16 struct {
123     Y uint16
124 }
125
126 func (c Gray16) RGBA() (r, g, b, a uint32) {
127     y := uint32(c.Y)
128     return y, y, y, 0xffff
129 }
130
131 // Model can convert any Color to one from its own color mod
132 // may be lossy.
133 type Model interface {
134     Convert(c Color) Color
135 }
136
137 // ModelFunc returns a Model that invokes f to implement the
138 func ModelFunc(f func(Color) Color) Model {
139     // Note: using *modelFunc as the implementation
140     // means that callers can still use comparisons

```

```

141         // like m == RGBAModel. This is not possible if
142         // we use the func value directly, because funcs
143         // are no longer comparable.
144         return &modelFunc{f}
145     }
146
147     type modelFunc struct {
148         f func(Color) Color
149     }
150
151     func (m *modelFunc) Convert(c Color) Color {
152         return m.f(c)
153     }
154
155     // Models for the standard color types.
156     var (
157         RGBAModel      Model = ModelFunc(rgbaModel)
158         RGBA64Model   Model = ModelFunc(rgba64Model)
159         NRGBAModel     Model = ModelFunc(nrgbaModel)
160         NRGBA64Model  Model = ModelFunc(nrgba64Model)
161         AlphaModel     Model = ModelFunc(alphaModel)
162         Alpha16Model  Model = ModelFunc(alpha16Model)
163         GrayModel      Model = ModelFunc(grayModel)
164         Gray16Model   Model = ModelFunc(gray16Model)
165     )
166
167     func rgbaModel(c Color) Color {
168         if _, ok := c.(RGBA); ok {
169             return c
170         }
171         r, g, b, a := c.RGBA()
172         return RGBA{uint8(r >> 8), uint8(g >> 8), uint8(b >>
173     }
174
175     func rgba64Model(c Color) Color {
176         if _, ok := c.(RGBA64); ok {
177             return c
178         }
179         r, g, b, a := c.RGBA()
180         return RGBA64{uint16(r), uint16(g), uint16(b), uint1
181     }
182
183     func nrgbaModel(c Color) Color {
184         if _, ok := c.(NRGBA); ok {
185             return c
186         }
187         r, g, b, a := c.RGBA()
188         if a == 0xffff {
189             return NRGBA{uint8(r >> 8), uint8(g >> 8), u

```

```

190     }
191     if a == 0 {
192         return NRGBA{0, 0, 0, 0}
193     }
194     // Since Color.RGBA returns a alpha-premultiplied co
195     r = (r * 0xffff) / a
196     g = (g * 0xffff) / a
197     b = (b * 0xffff) / a
198     return NRGBA{uint8(r >> 8), uint8(g >> 8), uint8(b >
199 }
200
201 func nrgba64Model(c Color) Color {
202     if _, ok := c.(NRGBA64); ok {
203         return c
204     }
205     r, g, b, a := c.RGBA()
206     if a == 0xffff {
207         return NRGBA64{uint16(r), uint16(g), uint16(
208     }
209     if a == 0 {
210         return NRGBA64{0, 0, 0, 0}
211     }
212     // Since Color.RGBA returns a alpha-premultiplied co
213     r = (r * 0xffff) / a
214     g = (g * 0xffff) / a
215     b = (b * 0xffff) / a
216     return NRGBA64{uint16(r), uint16(g), uint16(b), uint
217 }
218
219 func alphaModel(c Color) Color {
220     if _, ok := c.(Alpha); ok {
221         return c
222     }
223     _, _, _, a := c.RGBA()
224     return Alpha{uint8(a >> 8)}
225 }
226
227 func alpha16Model(c Color) Color {
228     if _, ok := c.(Alpha16); ok {
229         return c
230     }
231     _, _, _, a := c.RGBA()
232     return Alpha16{uint16(a)}
233 }
234
235 func grayModel(c Color) Color {
236     if _, ok := c.(Gray); ok {
237         return c
238     }
239     r, g, b, _ := c.RGBA()

```

```

240         y := (299*r + 587*g + 114*b + 500) / 1000
241         return Gray{uint8(y >> 8)}
242     }
243
244     func gray16Model(c Color) Color {
245         if _, ok := c.(Gray16); ok {
246             return c
247         }
248         r, g, b, _ := c.RGBA()
249         y := (299*r + 587*g + 114*b + 500) / 1000
250         return Gray16{uint16(y)}
251     }
252
253     // Palette is a palette of colors.
254     type Palette []Color
255
256     func diff(a, b uint32) uint32 {
257         if a > b {
258             return a - b
259         }
260         return b - a
261     }
262
263     // Convert returns the palette color closest to c in Euclidean
264     func (p Palette) Convert(c Color) Color {
265         if len(p) == 0 {
266             return nil
267         }
268         return p[p.Index(c)]
269     }
270
271     // Index returns the index of the palette color closest to c
272     // in R,G,B space.
273     func (p Palette) Index(c Color) int {
274         cr, cg, cb, _ := c.RGBA()
275         // Shift by 1 bit to avoid potential uint32 overflow
276         cr >>= 1
277         cg >>= 1
278         cb >>= 1
279         ret, bestSSD := 0, uint32(1<<32-1)
280         for i, v := range p {
281             vr, vg, vb, _ := v.RGBA()
282             vr >>= 1
283             vg >>= 1
284             vb >>= 1
285             dr, dg, db := diff(cr, vr), diff(cg, vg), diff(cb, vb)
286             ssd := (dr * dr) + (dg * dg) + (db * db)
287             if ssd < bestSSD {
288                 if ssd == 0 {

```

```
289             return i
290         }
291         ret, bestSSD = i, ssd
292     }
293 }
294 return ret
295 }
296
297 // Standard colors.
298 var (
299     Black      = Gray16{0}
300     White      = Gray16{0xffff}
301     Transparent = Alpha16{0}
302     Opaque     = Alpha16{0xffff}
303 )
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/image/color/ycbcr.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package color
6
7 // RGBToYCbCr converts an RGB triple to a Y'CbCr triple.
8 func RGBToYCbCr(r, g, b uint8) (uint8, uint8, uint8) {
9     // The JFIF specification says:
10    //     Y' = 0.2990*R + 0.5870*G + 0.1140*B
11    //     Cb = -0.1687*R - 0.3313*G + 0.5000*B + 128
12    //     Cr = 0.5000*R - 0.4187*G - 0.0813*B + 128
13    // http://www.w3.org/Graphics/JPEG/jfif3.pdf says Y
14    r1 := int(r)
15    g1 := int(g)
16    b1 := int(b)
17    yy := (19595*r1 + 38470*g1 + 7471*b1 + 1<<15) >> 16
18    cb := (-11056*r1 - 21712*g1 + 32768*b1 + 257<<15) >>
19    cr := (32768*r1 - 27440*g1 - 5328*b1 + 257<<15) >> 1
20    if yy < 0 {
21        yy = 0
22    } else if yy > 255 {
23        yy = 255
24    }
25    if cb < 0 {
26        cb = 0
27    } else if cb > 255 {
28        cb = 255
29    }
30    if cr < 0 {
31        cr = 0
32    } else if cr > 255 {
33        cr = 255
34    }
35    return uint8(yy), uint8(cb), uint8(cr)
36 }
37
38 // YCbCrToRGB converts a Y'CbCr triple to an RGB triple.
39 func YCbCrToRGB(y, cb, cr uint8) (uint8, uint8, uint8) {
40    // The JFIF specification says:
41    //     R = Y' + 1.40200*(Cr-128)
```

```

42         //      G = Y' - 0.34414*(Cb-128) - 0.71414*(Cr-128)
43         //      B = Y' + 1.77200*(Cb-128)
44         // http://www.w3.org/Graphics/JPEG/jfif3.pdf says Y
45         yy1 := int(y)<<16 + 1<<15
46         cb1 := int(cb) - 128
47         cr1 := int(cr) - 128
48         r := (yy1 + 91881*cr1) >> 16
49         g := (yy1 - 22554*cb1 - 46802*cr1) >> 16
50         b := (yy1 + 116130*cb1) >> 16
51         if r < 0 {
52             r = 0
53         } else if r > 255 {
54             r = 255
55         }
56         if g < 0 {
57             g = 0
58         } else if g > 255 {
59             g = 255
60         }
61         if b < 0 {
62             b = 0
63         } else if b > 255 {
64             b = 255
65         }
66         return uint8(r), uint8(g), uint8(b)
67     }
68
69     // YCbCr represents a fully opaque 24-bit Y'CbCr color, havi
70     // one luma and two chroma components.
71     //
72     // JPEG, VP8, the MPEG family and other codecs use this colo
73     // codecs often use the terms YUV and Y'CbCr interchangeably
74     // speaking, the term YUV applies only to analog video signa
75     // is Y (luminance) after applying gamma correction.
76     //
77     // Conversion between RGB and Y'CbCr is lossy and there are
78     // different formulae for converting between the two. This p
79     // the JFIF specification at http://www.w3.org/Graphics/JPEG
80     type YCbCr struct {
81         Y, Cb, Cr uint8
82     }
83
84     func (c YCbCr) RGBA() (uint32, uint32, uint32, uint32) {
85         r, g, b := YCbCrToRGB(c.Y, c.Cb, c.Cr)
86         return uint32(r) * 0x101, uint32(g) * 0x101, uint32(
87     }
88
89     // YCbCrModel is the Model for Y'CbCr colors.
90     var YCbCrModel Model = ModelFunc(yCbCrModel)
91

```

```
92 func yCbCrModel(c Color) Color {
93     if _, ok := c.(YCbCr); ok {
94         return c
95     }
96     r, g, b, _ := c.RGBA()
97     y, u, v := RGBToYCbCr(uint8(r>>8), uint8(g>>8), uint8(b>>8))
98     return YCbCr{y, u, v}
99 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/image/draw/draw.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package draw provides image composition functions.
6 //
7 // See "The Go image/draw package" for an introduction to th
8 // http://golang.org/doc/articles/image_draw.html
9 package draw
10
11 import (
12     "image"
13     "image/color"
14 )
15
16 // m is the maximum color value returned by image.Color.RGBA
17 const m = 1<<16 - 1
18
19 // Op is a Porter-Duff compositing operator.
20 type Op int
21
22 const (
23     // Over specifies ``src in mask) over dst''.
24     Over Op = iota
25     // Src specifies ``src in mask''.
26     Src
27 )
28
29 // A draw.Image is an image.Image with a Set method to chang
30 type Image interface {
31     image.Image
32     Set(x, y int, c color.Color)
33 }
34
35 // Draw calls DrawMask with a nil mask.
36 func Draw(dst Image, r image.Rectangle, src image.Image, sp
37     DrawMask(dst, r, src, sp, nil, image.ZP, op)
38 }
39
40 // clip clips r against each image's bounds (after translati
41 // destination image's co-ordinate space) and shifts the poi
```

```

42 // the same amount as the change in r.Min.
43 func clip(dst Image, r *image.Rectangle, src image.Image, sp
44     orig := r.Min
45     *r = r.Intersect(dst.Bounds())
46     *r = r.Intersect(src.Bounds().Add(orig.Sub(*sp)))
47     if mask != nil {
48         *r = r.Intersect(mask.Bounds().Add(orig.Sub(
49     }
50     dx := r.Min.X - orig.X
51     dy := r.Min.Y - orig.Y
52     if dx == 0 && dy == 0 {
53         return
54     }
55     (*sp).X += dx
56     (*sp).Y += dy
57     (*mp).X += dx
58     (*mp).Y += dy
59 }
60
61 // DrawMask aligns r.Min in dst with sp in src and mp in mas
62 // in dst with the result of a Porter-Duff composition. A ni
63 func DrawMask(dst Image, r image.Rectangle, src image.Image,
64     clip(dst, &r, src, &sp, mask, &mp)
65     if r.Empty() {
66         return
67     }
68
69     // Fast paths for special cases. If none of them app
70     if dst0, ok := dst.(*image.RGBA); ok {
71         if op == Over {
72             if mask == nil {
73                 switch src0 := src.(type) {
74                 case *image.Uniform:
75                     drawFillOver(dst0, r
76                     return
77                 case *image.RGBA:
78                     drawCopyOver(dst0, r
79                     return
80                 case *image.NRGBA:
81                     drawNRGBAOver(dst0,
82                     return
83                 case *image.YCbCr:
84                     drawYCbCr(dst0, r, s
85                     return
86             }
87         } else if mask0, ok := mask.(*image.
88             switch src0 := src.(type) {
89             case *image.Uniform:
90                 drawGlyphOver(dst0,
91                 return

```

```

92                                     }
93                                     }
94     } else {
95         if mask == nil {
96             switch src0 := src.(type) {
97             case *image.Uniform:
98                 drawFillSrc(dst0, r,
99                     return
100             case *image.RGBA:
101                 drawCopySrc(dst0, r,
102                     return
103             case *image.NRGBA:
104                 drawNRGBASrc(dst0, r
105                     return
106             case *image.YCbCr:
107                 drawYCbCr(dst0, r, s
108                     return
109             }
110         }
111     }
112     drawRGBA(dst0, r, src, sp, mask, mp, op)
113     return
114 }
115
116 x0, x1, dx := r.Min.X, r.Max.X, 1
117 y0, y1, dy := r.Min.Y, r.Max.Y, 1
118 if image.Image(dst) == src && r.Overlaps(r.Add(sp.Su
119     // Rectangles overlap: process backward?
120     if sp.Y < r.Min.Y || sp.Y == r.Min.Y && sp.X
121         x0, x1, dx = x1-1, x0-1, -1
122         y0, y1, dy = y1-1, y0-1, -1
123     }
124 }
125
126 var out *color.RGBA64
127 sy := sp.Y + y0 - r.Min.Y
128 my := mp.Y + y0 - r.Min.Y
129 for y := y0; y != y1; y, sy, my = y+dy, sy+dy, my+dy
130     sx := sp.X + x0 - r.Min.X
131     mx := mp.X + x0 - r.Min.X
132     for x := x0; x != x1; x, sx, mx = x+dx, sx+d
133         ma := uint32(m)
134         if mask != nil {
135             -, -, -, ma = mask.At(mx, my
136         }
137         switch {
138         case ma == 0:
139             if op == Over {
140                 // No-op.

```

```

141         } else {
142             dst.Set(x, y, color.
143         }
144     case ma == m && op == Src:
145         dst.Set(x, y, src.At(sx, sy)
146     default:
147         sr, sg, sb, sa := src.At(sx,
148         if out == nil {
149             out = new(color.RGBA
150         }
151         if op == Over {
152             dr, dg, db, da := ds
153             a := m - (sa * ma /
154             out.R = uint16((dr*a
155             out.G = uint16((dg*a
156             out.B = uint16((db*a
157             out.A = uint16((da*a
158         } else {
159             out.R = uint16(sr *
160             out.G = uint16(sg *
161             out.B = uint16(sb *
162             out.A = uint16(sa *
163         }
164         dst.Set(x, y, out)
165     }
166 }
167 }
168 }
169
170 func drawFillOver(dst *image.RGBA, r image.Rectangle, src *i
171     sr, sg, sb, sa := src.RGBA()
172     // The 0x101 is here for the same reason as in drawR
173     a := (m - sa) * 0x101
174     i0 := dst.PixOffset(r.Min.X, r.Min.Y)
175     i1 := i0 + r.Dx()*4
176     for y := r.Min.Y; y != r.Max.Y; y++ {
177         for i := i0; i < i1; i += 4 {
178             dr := uint32(dst.Pix[i+0])
179             dg := uint32(dst.Pix[i+1])
180             db := uint32(dst.Pix[i+2])
181             da := uint32(dst.Pix[i+3])
182
183             dst.Pix[i+0] = uint8((dr*a/m + sr) >
184             dst.Pix[i+1] = uint8((dg*a/m + sg) >
185             dst.Pix[i+2] = uint8((db*a/m + sb) >
186             dst.Pix[i+3] = uint8((da*a/m + sa) >
187         }
188         i0 += dst.Stride
189         i1 += dst.Stride

```

```

190     }
191 }
192
193 func drawFillSrc(dst *image.RGBA, r image.Rectangle, src *im
194     sr, sg, sb, sa := src.RGBA()
195     // The built-in copy function is faster than a strai
196     // the color, but copy requires a slice source. We t
197     // then use the first row as the slice source for th
198     i0 := dst.PixOffset(r.Min.X, r.Min.Y)
199     i1 := i0 + r.Dx()*4
200     for i := i0; i < i1; i += 4 {
201         dst.Pix[i+0] = uint8(sr >> 8)
202         dst.Pix[i+1] = uint8(sg >> 8)
203         dst.Pix[i+2] = uint8(sb >> 8)
204         dst.Pix[i+3] = uint8(sa >> 8)
205     }
206     firstRow := dst.Pix[i0:i1]
207     for y := r.Min.Y + 1; y < r.Max.Y; y++ {
208         i0 += dst.Stride
209         i1 += dst.Stride
210         copy(dst.Pix[i0:i1], firstRow)
211     }
212 }
213
214 func drawCopyOver(dst *image.RGBA, r image.Rectangle, src *i
215     dx, dy := r.Dx(), r.Dy()
216     d0 := dst.PixOffset(r.Min.X, r.Min.Y)
217     s0 := src.PixOffset(sp.X, sp.Y)
218     var (
219         ddelta, sdelta int
220         i0, i1, idelta int
221     )
222     if r.Min.Y < sp.Y || r.Min.Y == sp.Y && r.Min.X <= s
223         ddelta = dst.Stride
224         sdelta = src.Stride
225         i0, i1, idelta = 0, dx*4, +4
226     } else {
227         // If the source start point is higher than
228         // then we compose the rows in right-to-left
229         d0 += (dy - 1) * dst.Stride
230         s0 += (dy - 1) * src.Stride
231         ddelta = -dst.Stride
232         sdelta = -src.Stride
233         i0, i1, idelta = (dx-1)*4, -4, -4
234     }
235     for ; dy > 0; dy-- {
236         dpix := dst.Pix[d0:]
237         spix := src.Pix[s0:]
238         for i := i0; i != i1; i += idelta {
239             sr := uint32(spix[i+0]) * 0x101

```

```

240         sg := uint32(spix[i+1]) * 0x101
241         sb := uint32(spix[i+2]) * 0x101
242         sa := uint32(spix[i+3]) * 0x101
243
244         dr := uint32(dpix[i+0])
245         dg := uint32(dpix[i+1])
246         db := uint32(dpix[i+2])
247         da := uint32(dpix[i+3])
248
249         // The 0x101 is here for the same re
250         a := (m - sa) * 0x101
251
252         dpix[i+0] = uint8((dr*a/m + sr) >> 8)
253         dpix[i+1] = uint8((dg*a/m + sg) >> 8)
254         dpix[i+2] = uint8((db*a/m + sb) >> 8)
255         dpix[i+3] = uint8((da*a/m + sa) >> 8)
256     }
257     d0 += ddelta
258     s0 += sdelta
259 }
260 }
261
262 func drawCopySrc(dst *image.RGBA, r image.Rectangle, src *im
263     n, dy := 4*r.Dx(), r.Dy()
264     d0 := dst.PixOffset(r.Min.X, r.Min.Y)
265     s0 := src.PixOffset(sp.X, sp.Y)
266     var ddelta, sdelta int
267     if r.Min.Y <= sp.Y {
268         ddelta = dst.Stride
269         sdelta = src.Stride
270     } else {
271         // If the source start point is higher than
272         // in bottom-up order instead of top-down. U
273         // check the x co-ordinates because the buil
274         d0 += (dy - 1) * dst.Stride
275         s0 += (dy - 1) * src.Stride
276         ddelta = -dst.Stride
277         sdelta = -src.Stride
278     }
279     for ; dy > 0; dy-- {
280         copy(dst.Pix[d0:d0+n], src.Pix[s0:s0+n])
281         d0 += ddelta
282         s0 += sdelta
283     }
284 }
285
286 func drawNRGBAOver(dst *image.RGBA, r image.Rectangle, src *
287     i0 := (r.Min.X - dst.Rect.Min.X) * 4
288     i1 := (r.Max.X - dst.Rect.Min.X) * 4

```

```

289     si0 := (sp.X - src.Rect.Min.X) * 4
290     yMax := r.Max.Y - dst.Rect.Min.Y
291
292     y := r.Min.Y - dst.Rect.Min.Y
293     sy := sp.Y - src.Rect.Min.Y
294     for ; y != yMax; y, sy = y+1, sy+1 {
295         dpix := dst.Pix[y*dst.Stride:]
296         spix := src.Pix[sy*src.Stride:]
297
298         for i, si := i0, si0; i < i1; i, si = i+4, s
299             // Convert from non-premultiplied co
300             sa := uint32(spix[si+3]) * 0x101
301             sr := uint32(spix[si+0]) * sa / 0xff
302             sg := uint32(spix[si+1]) * sa / 0xff
303             sb := uint32(spix[si+2]) * sa / 0xff
304
305             dr := uint32(dpix[i+0])
306             dg := uint32(dpix[i+1])
307             db := uint32(dpix[i+2])
308             da := uint32(dpix[i+3])
309
310             // The 0x101 is here for the same re
311             a := (m - sa) * 0x101
312
313             dpix[i+0] = uint8((dr*a/m + sr) >> 8)
314             dpix[i+1] = uint8((dg*a/m + sg) >> 8)
315             dpix[i+2] = uint8((db*a/m + sb) >> 8)
316             dpix[i+3] = uint8((da*a/m + sa) >> 8)
317         }
318     }
319 }
320
321 func drawNRGBASrc(dst *image.RGBA, r image.Rectangle, src *i
322     i0 := (r.Min.X - dst.Rect.Min.X) * 4
323     i1 := (r.Max.X - dst.Rect.Min.X) * 4
324     si0 := (sp.X - src.Rect.Min.X) * 4
325     yMax := r.Max.Y - dst.Rect.Min.Y
326
327     y := r.Min.Y - dst.Rect.Min.Y
328     sy := sp.Y - src.Rect.Min.Y
329     for ; y != yMax; y, sy = y+1, sy+1 {
330         dpix := dst.Pix[y*dst.Stride:]
331         spix := src.Pix[sy*src.Stride:]
332
333         for i, si := i0, si0; i < i1; i, si = i+4, s
334             // Convert from non-premultiplied co
335             sa := uint32(spix[si+3]) * 0x101
336             sr := uint32(spix[si+0]) * sa / 0xff
337             sg := uint32(spix[si+1]) * sa / 0xff

```

```

338         sb := uint32(spix[si+2]) * sa / 0xff
339
340         dpix[i+0] = uint8(sr >> 8)
341         dpix[i+1] = uint8(sg >> 8)
342         dpix[i+2] = uint8(sb >> 8)
343         dpix[i+3] = uint8(sa >> 8)
344     }
345 }
346 }
347
348 func drawYCbCr(dst *image.RGBA, r image.Rectangle, src *image.YCbCr) {
349     // An image.YCbCr is always fully opaque, and so if
350     // (i.e. fully opaque) then the op is effectively all
351     x0 := (r.Min.X - dst.Rect.Min.X) * 4
352     x1 := (r.Max.X - dst.Rect.Min.X) * 4
353     y0 := r.Min.Y - dst.Rect.Min.Y
354     y1 := r.Max.Y - dst.Rect.Min.Y
355     switch src.SubsampleRatio {
356     case image.YCbCrSubsampleRatio422:
357         for y, sy := y0, sp.Y; y != y1; y, sy = y+1,
358             dpix := dst.Pix[y*dst.Stride:]
359             yi := (sy-src.Rect.Min.Y)*src.YStride
360             ciBase := (sy-src.Rect.Min.Y)*src.CStride
361             for x, sx := x0, sp.X; x != x1; x, s
362                 ci := ciBase + sx/2
363                 rr, gg, bb := color.YCbCrToRGB
364                 dpix[x+0] = rr
365                 dpix[x+1] = gg
366                 dpix[x+2] = bb
367                 dpix[x+3] = 255
368             }
369     }
370     case image.YCbCrSubsampleRatio420:
371         for y, sy := y0, sp.Y; y != y1; y, sy = y+1,
372             dpix := dst.Pix[y*dst.Stride:]
373             yi := (sy-src.Rect.Min.Y)*src.YStride
374             ciBase := (sy/2-src.Rect.Min.Y/2)*src.CStride
375             for x, sx := x0, sp.X; x != x1; x, s
376                 ci := ciBase + sx/2
377                 rr, gg, bb := color.YCbCrToRGB
378                 dpix[x+0] = rr
379                 dpix[x+1] = gg
380                 dpix[x+2] = bb
381                 dpix[x+3] = 255
382             }
383     }
384     default:
385         // Default to 4:4:4 subsampling.
386         for y, sy := y0, sp.Y; y != y1; y, sy = y+1,
387             dpix := dst.Pix[y*dst.Stride:]

```

```

388         yi := (sy-src.Rect.Min.Y)*src.YStride
389         ci := (sy-src.Rect.Min.Y)*src.CStride
390         for x := x0; x != x1; x, yi, ci = x+
391             rr, gg, bb := color.YCbCrToR
392                 dpix[x+0] = rr
393                 dpix[x+1] = gg
394                 dpix[x+2] = bb
395                 dpix[x+3] = 255
396             }
397         }
398     }
399 }
400
401 func drawGlyphOver(dst *image.RGBA, r image.Rectangle, src *
402     i0 := dst.PixOffset(r.Min.X, r.Min.Y)
403     i1 := i0 + r.Dx()*4
404     mi0 := mask.PixOffset(mp.X, mp.Y)
405     sr, sg, sb, sa := src.RGBA()
406     for y, my := r.Min.Y, mp.Y; y != r.Max.Y; y, my = y+
407         for i, mi := i0, mi0; i < i1; i, mi = i+4, r
408             ma := uint32(mask.Pix[mi])
409             if ma == 0 {
410                 continue
411             }
412             ma |= ma << 8
413
414             dr := uint32(dst.Pix[i+0])
415             dg := uint32(dst.Pix[i+1])
416             db := uint32(dst.Pix[i+2])
417             da := uint32(dst.Pix[i+3])
418
419             // The 0x101 is here for the same re
420             a := (m - (sa * ma / m)) * 0x101
421
422             dst.Pix[i+0] = uint8((dr*a + sr*ma)
423             dst.Pix[i+1] = uint8((dg*a + sg*ma)
424             dst.Pix[i+2] = uint8((db*a + sb*ma)
425             dst.Pix[i+3] = uint8((da*a + sa*ma)
426         }
427         i0 += dst.Stride
428         i1 += dst.Stride
429         mi0 += mask.Stride
430     }
431 }
432
433 func drawRGBA(dst *image.RGBA, r image.Rectangle, src image.
434     x0, x1, dx := r.Min.X, r.Max.X, 1
435     y0, y1, dy := r.Min.Y, r.Max.Y, 1
436     if image.Image(dst) == src && r.Overlaps(r.Add(sp.Su

```

```

437         if sp.Y < r.Min.Y || sp.Y == r.Min.Y && sp.X
438             x0, x1, dx = x1-1, x0-1, -1
439             y0, y1, dy = y1-1, y0-1, -1
440     }
441 }
442
443 sy := sp.Y + y0 - r.Min.Y
444 my := mp.Y + y0 - r.Min.Y
445 sx0 := sp.X + x0 - r.Min.X
446 mx0 := mp.X + x0 - r.Min.X
447 sx1 := sx0 + (x1 - x0)
448 i0 := dst.PixOffset(x0, y0)
449 di := dx * 4
450 for y := y0; y != y1; y, sy, my = y+dy, sy+dy, my+dy
451     for i, sx, mx := i0, sx0, mx0; sx != sx1; i,
452         ma := uint32(m)
453         if mask != nil {
454             -, -, -, ma = mask.At(mx, my)
455         }
456         sr, sg, sb, sa := src.At(sx, sy).RGB
457         if op == Over {
458             dr := uint32(dst.Pix[i+0])
459             dg := uint32(dst.Pix[i+1])
460             db := uint32(dst.Pix[i+2])
461             da := uint32(dst.Pix[i+3])
462
463             // dr, dg, db and da are all
464             // We work in 16-bit color,
465             // dr |= dr << 8
466             // and similarly for dg, db
467             // (which is a 16-bit color,
468             // This yields the same resu
469             a := (m - (sa * ma / m)) * 0
470
471             dst.Pix[i+0] = uint8((dr*a +
472             dst.Pix[i+1] = uint8((dg*a +
473             dst.Pix[i+2] = uint8((db*a +
474             dst.Pix[i+3] = uint8((da*a +
475
476         } else {
477             dst.Pix[i+0] = uint8(sr * ma
478             dst.Pix[i+1] = uint8(sg * ma
479             dst.Pix[i+2] = uint8(sb * ma
480             dst.Pix[i+3] = uint8(sa * ma
481         }
482     }
483     i0 += dy * dst.Stride
484 }
485 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/image/jpeg/fdct.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package jpeg
6
7 // This file implements a Forward Discrete Cosine Transform
8
9 /*
10 It is based on the code in jfdctint.c from the Independent JPEG
11 found at http://www.ijg.org/files/jpegsrc.v8c.tar.gz.
12
13 The "LEGAL ISSUES" section of the README in that archive say
14
15 In plain English:
16
17 1. We don't promise that this software works. (But if you find
18    please let us know!)
19 2. You can use this software for whatever you want. You don't
20 3. You may not pretend that you wrote this software. If you
21    program, you must acknowledge somewhere in your documentation
22    you've used the IJG code.
23
24 In legalese:
25
26 The authors make NO WARRANTY or representation, either express
27 with respect to this software, its quality, accuracy, merchantability,
28 fitness for a particular purpose. This software is provided "as is",
29 its user, assume the entire risk as to its quality and accuracy.
30
31 This software is copyright (C) 1991-2011, Thomas G. Lane, GNU
32 All Rights Reserved except as specified below.
33
34 Permission is hereby granted to use, copy, modify, and distribute
35 software (or portions thereof) for any purpose, without fee, under the
36 conditions:
37 (1) If any part of the source code for this software is distributed,
38 the README file must be included, with this copyright and no-warranty
39 unaltered; and any additions, deletions, or changes to the code
40 must be clearly indicated in accompanying documentation.
41 (2) If only executable code is distributed, then the accompanying
42 documentation must state that "this software is based in part on
43 the Independent JPEG Group".
44 (3) Permission for use of this software is granted only if the user
```

```

45 full responsibility for any undesirable consequences; the au
46 NO LIABILITY for damages of any kind.
47
48 These conditions apply to any software derived from or based
49 not just to the unmodified library. If you use our work, yo
50 acknowledge us.
51
52 Permission is NOT granted for the use of any IJG author's na
53 in advertising or publicity relating to this software or pro
54 it. This software may be referred to only as "the Independe
55 software".
56
57 We specifically permit and encourage the use of this softwar
58 commercial products, provided that all warranty or liability
59 assumed by the product vendor.
60 */
61
62 // Trigonometric constants in 13-bit fixed point format.
63 const (
64     fix_0_298631336 = 2446
65     fix_0_390180644 = 3196
66     fix_0_541196100 = 4433
67     fix_0_765366865 = 6270
68     fix_0_899976223 = 7373
69     fix_1_175875602 = 9633
70     fix_1_501321110 = 12299
71     fix_1_847759065 = 15137
72     fix_1_961570560 = 16069
73     fix_2_053119869 = 16819
74     fix_2_562915447 = 20995
75     fix_3_072711026 = 25172
76 )
77
78 const (
79     constBits      = 13
80     pass1Bits      = 2
81     centerJSample  = 128
82 )
83
84 // fdct performs a forward DCT on an 8x8 block of coefficient
85 // level shift.
86 func fdct(b *block) {
87     // Pass 1: process rows.
88     for y := 0; y < 8; y++ {
89         x0 := b[y*8+0]
90         x1 := b[y*8+1]
91         x2 := b[y*8+2]
92         x3 := b[y*8+3]
93         x4 := b[y*8+4]
94         x5 := b[y*8+5]

```

```

95         x6 := b[y*8+6]
96         x7 := b[y*8+7]
97
98         tmp0 := x0 + x7
99         tmp1 := x1 + x6
100        tmp2 := x2 + x5
101        tmp3 := x3 + x4
102
103        tmp10 := tmp0 + tmp3
104        tmp12 := tmp0 - tmp3
105        tmp11 := tmp1 + tmp2
106        tmp13 := tmp1 - tmp2
107
108        tmp0 = x0 - x7
109        tmp1 = x1 - x6
110        tmp2 = x2 - x5
111        tmp3 = x3 - x4
112
113        b[y*8+0] = (tmp10 + tmp11 - 8*centerJSample)
114        b[y*8+4] = (tmp10 - tmp11) << pass1Bits
115        z1 := (tmp12 + tmp13) * fix_0_541196100
116        z1 += 1 << (constBits - pass1Bits - 1)
117        b[y*8+2] = (z1 + tmp12*fix_0_765366865) >> (
118        b[y*8+6] = (z1 - tmp13*fix_1_847759065) >> (
119
120        tmp10 = tmp0 + tmp3
121        tmp11 = tmp1 + tmp2
122        tmp12 = tmp0 + tmp2
123        tmp13 = tmp1 + tmp3
124        z1 = (tmp12 + tmp13) * fix_1_175875602
125        z1 += 1 << (constBits - pass1Bits - 1)
126        tmp0 = tmp0 * fix_1_501321110
127        tmp1 = tmp1 * fix_3_072711026
128        tmp2 = tmp2 * fix_2_053119869
129        tmp3 = tmp3 * fix_0_298631336
130        tmp10 = tmp10 * -fix_0_899976223
131        tmp11 = tmp11 * -fix_2_562915447
132        tmp12 = tmp12 * -fix_0_390180644
133        tmp13 = tmp13 * -fix_1_961570560
134
135        tmp12 += z1
136        tmp13 += z1
137        b[y*8+1] = (tmp0 + tmp10 + tmp12) >> (constB
138        b[y*8+3] = (tmp1 + tmp11 + tmp13) >> (constB
139        b[y*8+5] = (tmp2 + tmp11 + tmp12) >> (constB
140        b[y*8+7] = (tmp3 + tmp10 + tmp13) >> (constB
141    }
142    // Pass 2: process columns.
143    // We remove pass1Bits scaling, but leave results sc

```

```

144     for x := 0; x < 8; x++ {
145         tmp0 := b[0*8+x] + b[7*8+x]
146         tmp1 := b[1*8+x] + b[6*8+x]
147         tmp2 := b[2*8+x] + b[5*8+x]
148         tmp3 := b[3*8+x] + b[4*8+x]
149
150         tmp10 := tmp0 + tmp3 + 1<<(pass1Bits-1)
151         tmp12 := tmp0 - tmp3
152         tmp11 := tmp1 + tmp2
153         tmp13 := tmp1 - tmp2
154
155         tmp0 = b[0*8+x] - b[7*8+x]
156         tmp1 = b[1*8+x] - b[6*8+x]
157         tmp2 = b[2*8+x] - b[5*8+x]
158         tmp3 = b[3*8+x] - b[4*8+x]
159
160         b[0*8+x] = (tmp10 + tmp11) >> pass1Bits
161         b[4*8+x] = (tmp10 - tmp11) >> pass1Bits
162
163         z1 := (tmp12 + tmp13) * fix_0_541196100
164         z1 += 1 << (constBits + pass1Bits - 1)
165         b[2*8+x] = (z1 + tmp12*fix_0_765366865) >> (
166         b[6*8+x] = (z1 - tmp13*fix_1_847759065) >> (
167
168         tmp10 = tmp0 + tmp3
169         tmp11 = tmp1 + tmp2
170         tmp12 = tmp0 + tmp2
171         tmp13 = tmp1 + tmp3
172         z1 = (tmp12 + tmp13) * fix_1_175875602
173         z1 += 1 << (constBits + pass1Bits - 1)
174         tmp0 = tmp0 * fix_1_501321110
175         tmp1 = tmp1 * fix_3_072711026
176         tmp2 = tmp2 * fix_2_053119869
177         tmp3 = tmp3 * fix_0_298631336
178         tmp10 = tmp10 * -fix_0_899976223
179         tmp11 = tmp11 * -fix_2_562915447
180         tmp12 = tmp12 * -fix_0_390180644
181         tmp13 = tmp13 * -fix_1_961570560
182
183         tmp12 += z1
184         tmp13 += z1
185         b[1*8+x] = (tmp0 + tmp10 + tmp12) >> (constB
186         b[3*8+x] = (tmp1 + tmp11 + tmp13) >> (constB
187         b[5*8+x] = (tmp2 + tmp11 + tmp12) >> (constB
188         b[7*8+x] = (tmp3 + tmp10 + tmp13) >> (constB
189     }
190 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/image/jpeg/huffman.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package jpeg
6
7 import "io"
8
9 // Each code is at most 16 bits long.
10 const maxCodeLength = 16
11
12 // Each decoded value is a uint8, so there are at most 256 s
13 const maxNumValues = 256
14
15 // Bit stream for the Huffman decoder.
16 // The n least significant bits of a form the unread bits, t
17 type bits struct {
18     a int // accumulator.
19     n int // the number of unread bits in a.
20     m int // mask. m==1<<(n-1) when n>0, with m==0 when
21 }
22
23 // Huffman table decoder, specified in section C.
24 type huffman struct {
25     l [maxCodeLength]int
26     length int // sum of l[i].
27     val [maxNumValues]uint8 // the decoded values,
28     size [maxNumValues]int // size[i] is the numbe
29     code [maxNumValues]int // code[i] is the encod
30     minCode [maxCodeLength]int // min codes of length
31     maxCode [maxCodeLength]int // max codes of length
32     valIndex [maxCodeLength]int // index into val of mi
33 }
34
35 // Reads bytes from the io.Reader to ensure that bits.n is a
36 func (d *decoder) ensureNBits(n int) error {
37     for d.b.n < n {
38         c, err := d.r.ReadByte()
39         if err != nil {
40             return err
41         }
42     }
43 }
```

```

42         d.b.a = d.b.a<<8 | int(c)
43         d.b.n += 8
44         if d.b.m == 0 {
45             d.b.m = 1 << 7
46         } else {
47             d.b.m <=& 8
48         }
49         // Byte stuffing, specified in section F.1.2
50         if c == 0xff {
51             c, err = d.r.ReadByte()
52             if err != nil {
53                 return err
54             }
55             if c != 0x00 {
56                 return FormatError("missing
57             }
58         }
59     }
60     return nil
61 }
62
63 // The composition of RECEIVE and EXTEND, specified in secti
64 func (d *decoder) receiveExtend(t uint8) (int, error) {
65     err := d.ensureNBits(int(t))
66     if err != nil {
67         return 0, err
68     }
69     d.b.n -= int(t)
70     d.b.m >>= t
71     s := 1 << t
72     x := (d.b.a >> uint8(d.b.n)) & (s - 1)
73     if x < s>>1 {
74         x += ((-1) << t) + 1
75     }
76     return x, nil
77 }
78
79 // Processes a Define Huffman Table marker, and initializes
80 // Specified in section B.2.4.2.
81 func (d *decoder) processDHT(n int) error {
82     for n > 0 {
83         if n < 17 {
84             return FormatError("DHT has wrong le
85         }
86         _, err := io.ReadFull(d.r, d.tmp[0:17])
87         if err != nil {
88             return err
89         }
90         tc := d.tmp[0] >> 4
91         if tc > maxTc {

```

```

92         return FormatError("bad Tc value")
93     }
94     th := d.tmp[0] & 0x0f
95     const isBaseline = true // Progressive mode
96     if th > maxTh || isBaseline && th > 1 {
97         return FormatError("bad Th value")
98     }
99     h := &d.huff[tc][th]
100
101     // Read l and val (and derive length).
102     h.length = 0
103     for i := 0; i < maxCodeLength; i++ {
104         h.l[i] = int(d.tmp[i+1])
105         h.length += h.l[i]
106     }
107     if h.length == 0 {
108         return FormatError("Huffman table ha
109     }
110     if h.length > maxNumValues {
111         return FormatError("Huffman table ha
112     }
113     n -= h.length + 17
114     if n < 0 {
115         return FormatError("DHT has wrong le
116     }
117     _, err = io.ReadFull(d.r, h.val[0:h.length])
118     if err != nil {
119         return err
120     }
121
122     // Derive size.
123     k := 0
124     for i := 0; i < maxCodeLength; i++ {
125         for j := 0; j < h.l[i]; j++ {
126             h.size[k] = i + 1
127             k++
128         }
129     }
130
131     // Derive code.
132     code := 0
133     size := h.size[0]
134     for i := 0; i < h.length; i++ {
135         if size != h.size[i] {
136             code <<= uint8(h.size[i] - s
137             size = h.size[i]
138         }
139         h.code[i] = code
140         code++

```

```

141     }
142
143     // Derive minCode, maxCode, and valIndex.
144     k = 0
145     index := 0
146     for i := 0; i < maxCodeLength; i++ {
147         if h.l[i] == 0 {
148             h.minCode[i] = -1
149             h.maxCode[i] = -1
150             h.valIndex[i] = -1
151         } else {
152             h.minCode[i] = k
153             h.maxCode[i] = k + h.l[i] -
154             h.valIndex[i] = index
155             k += h.l[i]
156             index += h.l[i]
157         }
158         k <<= 1
159     }
160 }
161 return nil
162 }
163
164 // Returns the next Huffman-coded value from the bit stream,
165 // TODO(nigeltao): This decoding algorithm is simple, but sl
166 // peeling off only 1 bit at a time, ought to be faster.
167 func (d *decoder) decodeHuffman(h *huffman) (uint8, error) {
168     if h.length == 0 {
169         return 0, FormatError("uninitialized Huffman
170     }
171     for i, code := 0, 0; i < maxCodeLength; i++ {
172         err := d.ensureNBits(1)
173         if err != nil {
174             return 0, err
175         }
176         if d.b.a&d.b.m != 0 {
177             code |= 1
178         }
179         d.b.n--
180         d.b.m >>= 1
181         if code <= h.maxCode[i] {
182             return h.val[h.valIndex[i]+code-h.mi
183         }
184         code <<= 1
185     }
186     return 0, FormatError("bad Huffman code")
187 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/image/jpeg/idct.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package jpeg
6
7 // This is a Go translation of idct.c from
8 //
9 // http://standards.iso.org/ittf/PubliclyAvailableStandards/
10 //
11 // which carries the following notice:
12
13 /* Copyright (C) 1996, MPEG Software Simulation Group. All R
14
15 /*
16  * Disclaimer of Warranty
17  *
18  * These software programs are available to the user without
19  * royalty on an "as is" basis. The MPEG Software Simulatio
20  * any and all warranties, whether express, implied, or stat
21  * implied warranties or merchantability or of fitness for a
22  * purpose. In no event shall the copyright-holder be liabl
23  * incidental, punitive, or consequential damages of any kin
24  * arising from the use of these programs.
25  *
26  * This disclaimer of warranty extends to the user of these
27  * customers, employees, agents, transferees, successors, an
28  *
29  * The MPEG Software Simulation Group does not represent or
30  * programs furnished hereunder are free of infringement of
31  * patents.
32  *
33  * Commercial implementations of MPEG-1 and MPEG-2 video, in
34  * are subject to royalty fees to patent holders. Many of t
35  * general enough such that they are unavoidable regardless
36  * design.
37  *
38  */
39
40 const (
41     w1 = 2841 // 2048*sqrt(2)*cos(1*pi/16)
42     w2 = 2676 // 2048*sqrt(2)*cos(2*pi/16)
43     w3 = 2408 // 2048*sqrt(2)*cos(3*pi/16)
44     w5 = 1609 // 2048*sqrt(2)*cos(5*pi/16)
```

```

45         w6 = 1108 // 2048*sqrt(2)*cos(6*pi/16)
46         w7 = 565  // 2048*sqrt(2)*cos(7*pi/16)
47
48         w1pw7 = w1 + w7
49         w1mw7 = w1 - w7
50         w2pw6 = w2 + w6
51         w2mw6 = w2 - w6
52         w3pw5 = w3 + w5
53         w3mw5 = w3 - w5
54
55         r2 = 181 // 256/sqrt(2)
56     )
57
58 // idct performs a 2-D Inverse Discrete Cosine Transformatio
59 // +128 level shift and a clip to [0, 255], writing the resu
60 // stride is the number of elements between successive rows
61 //
62 // The input coefficients should already have been multiplie
63 // appropriate quantization table. We use fixed-point comput
64 // number of bits for the fractional component varying over
65 // stages.
66 //
67 // For more on the actual algorithm, see Z. Wang, "Fast algo
68 // discrete W transform and for the discrete Fourier transfo
69 // ASSP, Vol. ASSP- 32, pp. 803-816, Aug. 1984.
70 func idct(dst []byte, stride int, src *block) {
71     // Horizontal 1-D IDCT.
72     for y := 0; y < 8; y++ {
73         // If all the AC components are zero, then t
74         if src[y*8+1] == 0 && src[y*8+2] == 0 && src
75             src[y*8+4] == 0 && src[y*8+5] == 0 &
76             dc := src[y*8+0] << 3
77             src[y*8+0] = dc
78             src[y*8+1] = dc
79             src[y*8+2] = dc
80             src[y*8+3] = dc
81             src[y*8+4] = dc
82             src[y*8+5] = dc
83             src[y*8+6] = dc
84             src[y*8+7] = dc
85             continue
86     }
87
88     // Prescale.
89     x0 := (src[y*8+0] << 11) + 128
90     x1 := src[y*8+4] << 11
91     x2 := src[y*8+6]
92     x3 := src[y*8+2]
93     x4 := src[y*8+1]
94     x5 := src[y*8+7]

```

```

95         x6 := src[y*8+5]
96         x7 := src[y*8+3]
97
98         // Stage 1.
99         x8 := w7 * (x4 + x5)
100        x4 = x8 + w1mw7*x4
101        x5 = x8 - w1pw7*x5
102        x8 = w3 * (x6 + x7)
103        x6 = x8 - w3mw5*x6
104        x7 = x8 - w3pw5*x7
105
106        // Stage 2.
107        x8 = x0 + x1
108        x0 -= x1
109        x1 = w6 * (x3 + x2)
110        x2 = x1 - w2pw6*x2
111        x3 = x1 + w2mw6*x3
112        x1 = x4 + x6
113        x4 -= x6
114        x6 = x5 + x7
115        x5 -= x7
116
117        // Stage 3.
118        x7 = x8 + x3
119        x8 -= x3
120        x3 = x0 + x2
121        x0 -= x2
122        x2 = (r2*(x4+x5) + 128) >> 8
123        x4 = (r2*(x4-x5) + 128) >> 8
124
125        // Stage 4.
126        src[8*y+0] = (x7 + x1) >> 8
127        src[8*y+1] = (x3 + x2) >> 8
128        src[8*y+2] = (x0 + x4) >> 8
129        src[8*y+3] = (x8 + x6) >> 8
130        src[8*y+4] = (x8 - x6) >> 8
131        src[8*y+5] = (x0 - x4) >> 8
132        src[8*y+6] = (x3 - x2) >> 8
133        src[8*y+7] = (x7 - x1) >> 8
134    }
135
136    // Vertical 1-D IDCT.
137    for x := 0; x < 8; x++ {
138        // Similar to the horizontal 1-D IDCT case,
139        // However, after performing the horizontal
140        // we do not bother to check for the all-zero
141
142        // Prescale.
143        y0 := (src[8*0+x] << 8) + 8192

```

```

144     y1 := src[8*4+x] << 8
145     y2 := src[8*6+x]
146     y3 := src[8*2+x]
147     y4 := src[8*1+x]
148     y5 := src[8*7+x]
149     y6 := src[8*5+x]
150     y7 := src[8*3+x]
151
152     // Stage 1.
153     y8 := w7*(y4+y5) + 4
154     y4 = (y8 + w1mw7*y4) >> 3
155     y5 = (y8 - w1pw7*y5) >> 3
156     y8 = w3*(y6+y7) + 4
157     y6 = (y8 - w3mw5*y6) >> 3
158     y7 = (y8 - w3pw5*y7) >> 3
159
160     // Stage 2.
161     y8 = y0 + y1
162     y0 -= y1
163     y1 = w6*(y3+y2) + 4
164     y2 = (y1 - w2pw6*y2) >> 3
165     y3 = (y1 + w2mw6*y3) >> 3
166     y1 = y4 + y6
167     y4 -= y6
168     y6 = y5 + y7
169     y5 -= y7
170
171     // Stage 3.
172     y7 = y8 + y3
173     y8 -= y3
174     y3 = y0 + y2
175     y0 -= y2
176     y2 = (r2*(y4+y5) + 128) >> 8
177     y4 = (r2*(y4-y5) + 128) >> 8
178
179     // Stage 4.
180     src[8*0+x] = (y7 + y1) >> 14
181     src[8*1+x] = (y3 + y2) >> 14
182     src[8*2+x] = (y0 + y4) >> 14
183     src[8*3+x] = (y8 + y6) >> 14
184     src[8*4+x] = (y8 - y6) >> 14
185     src[8*5+x] = (y0 - y4) >> 14
186     src[8*6+x] = (y3 - y2) >> 14
187     src[8*7+x] = (y7 - y1) >> 14
188 }
189
190 // Level shift by +128, clip to [0, 255], and write
191 for y := 0; y < 8; y++ {
192     for x := 0; x < 8; x++ {

```

```
193         c := src[y*8+x]
194         if c < -128 {
195             c = 0
196         } else if c > 127 {
197             c = 255
198         } else {
199             c += 128
200         }
201         dst[y*stride+x] = uint8(c)
202     }
203 }
204 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/image/jpeg/reader.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package jpeg implements a JPEG image decoder and encoder.
6 //
7 // JPEG is defined in ITU-T T.81: http://www.w3.org/Graphics
8 package jpeg
9
10 import (
11     "bufio"
12     "image"
13     "image/color"
14     "io"
15 )
16
17 // TODO(nigeltao): fix up the doc comment style so that sent
18 // the name of the type or function that they annotate.
19
20 // A FormatError reports that the input is not a valid JPEG.
21 type FormatError string
22
23 func (e FormatError) Error() string { return "invalid JPEG f"
24
25 // An UnsupportedError reports that the input uses a valid b
26 type UnsupportedError string
27
28 func (e UnsupportedError) Error() string { return "unsupport
29
30 // Component specification, specified in section B.2.2.
31 type component struct {
32     h int // Horizontal sampling factor.
33     v int // Vertical sampling factor.
34     c uint8 // Component identifier.
35     tq uint8 // Quantization table destination selector.
36 }
37
38 type block [blockSize]int
39
40 const (
41     blockSize = 64 // A DCT block is 8x8.
```

```

42
43     dcTable = 0
44     acTable = 1
45     maxTc   = 1
46     maxTh   = 3
47     maxTq   = 3
48
49     // A grayscale JPEG image has only a Y component.
50     nGrayComponent = 1
51     // A color JPEG image has Y, Cb and Cr components.
52     nColorComponent = 3
53
54     // We only support 4:4:4, 4:2:2 and 4:2:0 downsampli
55     // number of luma samples per chroma sample is at mo
56     // and 2 in the vertical direction.
57     maxH = 2
58     maxV = 2
59 )
60
61 const (
62     soiMarker    = 0xd8 // Start Of Image.
63     eoiMarker    = 0xd9 // End Of Image.
64     sof0Marker   = 0xc0 // Start Of Frame (Baseline).
65     sof2Marker   = 0xc2 // Start Of Frame (Progressive).
66     dhtMarker    = 0xc4 // Define Huffman Table.
67     dqtMarker    = 0xdb // Define Quantization Table.
68     sosMarker    = 0xda // Start Of Scan.
69     driMarker    = 0xdd // Define Restart Interval.
70     rst0Marker   = 0xd0 // ReStArt (0).
71     rst7Marker   = 0xd7 // ReStArt (7).
72     app0Marker   = 0xe0 // APPLication specific (0).
73     app15Marker  = 0xef // APPLication specific (15).
74     comMarker    = 0xfe // COMment.
75 )
76
77 // Maps from the zig-zag ordering to the natural ordering.
78 var unzig = [blockSize]int{
79     0, 1, 8, 16, 9, 2, 3, 10,
80     17, 24, 32, 25, 18, 11, 4, 5,
81     12, 19, 26, 33, 40, 48, 41, 34,
82     27, 20, 13, 6, 7, 14, 21, 28,
83     35, 42, 49, 56, 57, 50, 43, 36,
84     29, 22, 15, 23, 30, 37, 44, 51,
85     58, 59, 52, 45, 38, 31, 39, 46,
86     53, 60, 61, 54, 47, 55, 62, 63,
87 }
88
89 // If the passed in io.Reader does not also have ReadByte, t
90 type Reader interface {
91     io.Reader

```

```

92         ReadByte() (c byte, err error)
93     }
94
95     type decoder struct {
96         r          Reader
97         width, height int
98         img1       *image.Gray
99         img3       *image.YCbCr
100        ri        int // Restart Interval.
101        nComp      int
102        comp       [nColorComponent]component
103        huff       [maxTc + 1][maxTh + 1]huffman
104        quant      [maxTq + 1]block
105        b          bits
106        tmp        [1024]byte
107    }
108
109    // Reads and ignores the next n bytes.
110    func (d *decoder) ignore(n int) error {
111        for n > 0 {
112            m := len(d.tmp)
113            if m > n {
114                m = n
115            }
116            _, err := io.ReadFull(d.r, d.tmp[0:m])
117            if err != nil {
118                return err
119            }
120            n -= m
121        }
122        return nil
123    }
124
125    // Specified in section B.2.2.
126    func (d *decoder) processSOF(n int) error {
127        switch n {
128        case 6 + 3*nGrayComponent:
129            d.nComp = nGrayComponent
130        case 6 + 3*nColorComponent:
131            d.nComp = nColorComponent
132        default:
133            return UnsupportedError("SOF has wrong lengt
134        }
135        _, err := io.ReadFull(d.r, d.tmp[:n])
136        if err != nil {
137            return err
138        }
139        // We only support 8-bit precision.
140        if d.tmp[0] != 8 {

```

```

141         return UnsupportedError("precision")
142     }
143     d.height = int(d.tmp[1])<<8 + int(d.tmp[2])
144     d.width = int(d.tmp[3])<<8 + int(d.tmp[4])
145     if int(d.tmp[5]) != d.nComp {
146         return UnsupportedError("SOF has wrong numbe
147     }
148     for i := 0; i < d.nComp; i++ {
149         hv := d.tmp[7+3*i]
150         d.comp[i].h = int(hv >> 4)
151         d.comp[i].v = int(hv & 0x0f)
152         d.comp[i].c = d.tmp[6+3*i]
153         d.comp[i].tq = d.tmp[8+3*i]
154         if d.nComp == nGrayComponent {
155             continue
156         }
157         // For color images, we only support 4:4:4,
158         // downsampling ratios. This implies that th
159         // component are either (1, 1), (2, 1) or (2
160         // values for the Cr and Cb components must
161         if i == 0 {
162             if hv != 0x11 && hv != 0x21 && hv !=
163                 return UnsupportedError("lum
164             }
165         } else if hv != 0x11 {
166             return UnsupportedError("chroma down
167         }
168     }
169     return nil
170 }
171
172 // Specified in section B.2.4.1.
173 func (d *decoder) processDQT(n int) error {
174     const qtLength = 1 + blockSize
175     for ; n >= qtLength; n -= qtLength {
176         _, err := io.ReadFull(d.r, d.tmp[0:qtLength])
177         if err != nil {
178             return err
179         }
180         pq := d.tmp[0] >> 4
181         if pq != 0 {
182             return UnsupportedError("bad Pq valu
183         }
184         tq := d.tmp[0] & 0x0f
185         if tq > maxTq {
186             return FormatError("bad Tq value")
187         }
188         for i := range d.quant[tq] {
189             d.quant[tq][i] = int(d.tmp[i+1])

```

```

190         }
191     }
192     if n != 0 {
193         return FormatError("DQT has wrong length")
194     }
195     return nil
196 }
197
198 // makeImg allocates and initializes the destination image.
199 func (d *decoder) makeImg(h0, v0, mxx, myy int) {
200     if d.nComp == nGrayComponent {
201         m := image.NewGray(image.Rect(0, 0, 8*mxx, 8
202             d.img1 = m.SubImage(image.Rect(0, 0, d.width
203             return
204     }
205     var subsampleRatio image.YCbCrSubsampleRatio
206     switch h0 * v0 {
207     case 1:
208         subsampleRatio = image.YCbCrSubsampleRatio44
209     case 2:
210         subsampleRatio = image.YCbCrSubsampleRatio42
211     case 4:
212         subsampleRatio = image.YCbCrSubsampleRatio42
213     default:
214         panic("unreachable")
215     }
216     m := image.NewYCbCr(image.Rect(0, 0, 8*h0*mxx, 8*v0*
217         d.img3 = m.SubImage(image.Rect(0, 0, d.width, d.heig
218 }
219
220 // Specified in section B.2.3.
221 func (d *decoder) processSOS(n int) error {
222     if d.nComp == 0 {
223         return FormatError("missing SOF marker")
224     }
225     if n != 4+2*d.nComp {
226         return UnsupportedError("SOS has wrong lengt
227     }
228     _, err := io.ReadFull(d.r, d.tmp[0:4+2*d.nComp])
229     if err != nil {
230         return err
231     }
232     if int(d.tmp[0]) != d.nComp {
233         return UnsupportedError("SOS has wrong numbe
234     }
235     var scan [nColorComponent]struct {
236         td uint8 // DC table selector.
237         ta uint8 // AC table selector.
238     }
239     for i := 0; i < d.nComp; i++ {

```

```

240         cs := d.tmp[1+2*i] // Component selector.
241         if cs != d.comp[i].c {
242             return UnsupportedError("scan compon
243         }
244         scan[i].td = d.tmp[2+2*i] >> 4
245         scan[i].ta = d.tmp[2+2*i] & 0x0f
246     }
247     // mxx and myy are the number of MCUs (Minimum Coded
248     h0, v0 := d.comp[0].h, d.comp[0].v // The h and v va
249     mxx := (d.width + 8*h0 - 1) / (8 * h0)
250     myy := (d.height + 8*v0 - 1) / (8 * v0)
251     if d.img1 == nil && d.img3 == nil {
252         d.makeImg(h0, v0, mxx, myy)
253     }
254
255     mcu, expectedRST := 0, uint8(rst0Marker)
256     var (
257         b block
258         dc [nColorComponent]int
259     )
260     for my := 0; my < myy; my++ {
261         for mx := 0; mx < mxx; mx++ {
262             for i := 0; i < d.nComp; i++ {
263                 qt := &d.quant[d.comp[i].tq]
264                 for j := 0; j < d.comp[i].h*
265                     // TODO(nigeltao): r
266                     // analysis is good
267                     b = block{}
268
269                 // Decode the DC coe
270                 value, err := d.deco
271                 if err != nil {
272                     return err
273                 }
274                 if value > 16 {
275                     return Unsup
276                 }
277                 dcDelta, err := d.re
278                 if err != nil {
279                     return err
280                 }
281                 dc[i] += dcDelta
282                 b[0] = dc[i] * qt[0]
283
284                 // Decode the AC coe
285                 for k := 1; k < bloc
286                     value, err :
287                     if err != ni
288                     retu

```

```

289     }
290     val0 := valu
291     val1 := valu
292     if val1 != 0
293         k +=
294         if k
295
296     }
297     ac,
298     if e
299
300     }
301     b[un
302 } else {
303     if v
304
305     }
306     k +=
307 }
308 }
309
310 // Perform the inver
311 if d.nComp == nGrayC
312     idct(d.img1.
313 } else {
314     switch i {
315     case 0:
316         mx0
317         my0
318         idct
319     case 1:
320         idct
321     case 2:
322         idct
323     }
324     }
325     } // for j
326 } // for i
327 mcu++
328 if d.ri > 0 && mcu%d.ri == 0 && mcu
329     // A more sophisticated deco
330     // but this one assumes well
331     _, err := io.ReadFull(d.r, d
332     if err != nil {
333         return err
334     }
335     if d.tmp[0] != 0xff || d.tmp
336         return FormatError("
337 }

```

```

338             expectedRST++
339             if expectedRST == rst7Marker
340                 expectedRST = rst0Ma
341             }
342             // Reset the Huffman decoder
343             d.b = bits{}
344             // Reset the DC components,
345             dc = [nColorComponent]int{}
346         }
347     } // for mx
348 } // for my
349
350     return nil
351 }
352
353 // Specified in section B.2.4.4.
354 func (d *decoder) processDRI(n int) error {
355     if n != 2 {
356         return FormatError("DRI has wrong length")
357     }
358     _, err := io.ReadFull(d.r, d.tmp[0:2])
359     if err != nil {
360         return err
361     }
362     d.ri = int(d.tmp[0])<<8 + int(d.tmp[1])
363     return nil
364 }
365
366 // decode reads a JPEG image from r and returns it as an ima
367 func (d *decoder) decode(r io.Reader, configOnly bool) (imag
368     if rr, ok := r.(Reader); ok {
369         d.r = rr
370     } else {
371         d.r = bufio.NewReader(r)
372     }
373
374     // Check for the Start Of Image marker.
375     _, err := io.ReadFull(d.r, d.tmp[0:2])
376     if err != nil {
377         return nil, err
378     }
379     if d.tmp[0] != 0xff || d.tmp[1] != soiMarker {
380         return nil, FormatError("missing SOI marker")
381     }
382
383     // Process the remaining segments until the End Of I
384     for {
385         _, err := io.ReadFull(d.r, d.tmp[0:2])
386         if err != nil {
387             return nil, err

```

```

388     }
389     if d.tmp[0] != 0xff {
390         return nil, FormatError("missing 0xf
391     }
392     marker := d.tmp[1]
393     if marker == eoiMarker { // End Of Image.
394         break
395     }
396
397     // Read the 16-bit length of the segment. Th
398     // length itself, so we subtract 2 to get th
399     _, err = io.ReadFull(d.r, d.tmp[0:2])
400     if err != nil {
401         return nil, err
402     }
403     n := int(d.tmp[0])<<8 + int(d.tmp[1]) - 2
404     if n < 0 {
405         return nil, FormatError("short segme
406     }
407
408     switch {
409     case marker == sof0Marker: // Start Of Frame
410         err = d.processSOF(n)
411         if configOnly {
412             return nil, err
413         }
414     case marker == sof2Marker: // Start Of Frame
415         err = UnsupportedError("progressive
416     case marker == dhtMarker: // Define Huffman
417         err = d.processDHT(n)
418     case marker == dqtMarker: // Define Quantiza
419         err = d.processDQT(n)
420     case marker == sosMarker: // Start Of Scan.
421         err = d.processSOS(n)
422     case marker == driMarker: // Define Restart
423         err = d.processDRI(n)
424     case marker >= app0Marker && marker <= app15
425         err = d.ignore(n)
426     default:
427         err = UnsupportedError("unknown mark
428     }
429     if err != nil {
430         return nil, err
431     }
432 }
433 if d.img1 != nil {
434     return d.img1, nil
435 }
436 if d.img3 != nil {

```

```

437         return d.img3, nil
438     }
439     return nil, FormatError("missing SOS marker")
440 }
441
442 // Decode reads a JPEG image from r and returns it as an ima
443 func Decode(r io.Reader) (image.Image, error) {
444     var d decoder
445     return d.decode(r, false)
446 }
447
448 // DecodeConfig returns the color model and dimensions of a
449 // decoding the entire image.
450 func DecodeConfig(r io.Reader) (image.Config, error) {
451     var d decoder
452     if _, err := d.decode(r, true); err != nil {
453         return image.Config{}, err
454     }
455     switch d.nComp {
456     case nGrayComponent:
457         return image.Config{
458             ColorModel: color.GrayModel,
459             Width:       d.width,
460             Height:    d.height,
461         }, nil
462     case nColorComponent:
463         return image.Config{
464             ColorModel: color.YCbCrModel,
465             Width:       d.width,
466             Height:    d.height,
467         }, nil
468     }
469     return image.Config{}, FormatError("missing SOF mark
470 }
471
472 func init() {
473     image.RegisterFormat("jpeg", "\xff\xd8", Decode, Dec
474 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/image/jpeg/writer.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package jpeg
6
7 import (
8     "bufio"
9     "errors"
10    "image"
11    "image/color"
12    "io"
13 )
14
15 // min returns the minimum of two integers.
16 func min(x, y int) int {
17     if x < y {
18         return x
19     }
20     return y
21 }
22
23 // div returns a/b rounded to the nearest integer, instead of
24 func div(a int, b int) int {
25     if a >= 0 {
26         return (a + (b >> 1)) / b
27     }
28     return -((-a + (b >> 1)) / b)
29 }
30
31 // bitCount counts the number of bits needed to hold an integer
32 var bitCount = [256]byte{
33     0, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4,
34     5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
35     6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
36     6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,
37     7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
38     7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
39     7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
40     7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
41     8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
```

```

42         8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
43         8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
44         8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
45         8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
46         8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
47         8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
48         8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
49     }
50
51     type quantIndex int
52
53     const (
54         quantIndexLuminance quantIndex = iota
55         quantIndexChrominance
56         nQuantIndex
57     )
58
59     // unscaledQuant are the unscaled quantization tables. Each
60     // scales the tables according to its quality parameter.
61     var unscaledQuant = [nQuantIndex][blockSize]byte{
62         // Luminance.
63         {
64             16, 11, 10, 16, 24, 40, 51, 61,
65             12, 12, 14, 19, 26, 58, 60, 55,
66             14, 13, 16, 24, 40, 57, 69, 56,
67             14, 17, 22, 29, 51, 87, 80, 62,
68             18, 22, 37, 56, 68, 109, 103, 77,
69             24, 35, 55, 64, 81, 104, 113, 92,
70             49, 64, 78, 87, 103, 121, 120, 101,
71             72, 92, 95, 98, 112, 100, 103, 99,
72         },
73         // Chrominance.
74         {
75             17, 18, 24, 47, 99, 99, 99, 99,
76             18, 21, 26, 66, 99, 99, 99, 99,
77             24, 26, 56, 99, 99, 99, 99, 99,
78             47, 66, 99, 99, 99, 99, 99, 99,
79             99, 99, 99, 99, 99, 99, 99, 99,
80             99, 99, 99, 99, 99, 99, 99, 99,
81             99, 99, 99, 99, 99, 99, 99, 99,
82             99, 99, 99, 99, 99, 99, 99, 99,
83         },
84     }
85
86     type huffIndex int
87
88     const (
89         huffIndexLuminanceDC huffIndex = iota
90         huffIndexLuminanceAC
91         huffIndexChrominanceDC

```

```

92         huffIndexChrominanceAC
93         nHuffIndex
94     )
95
96     // huffmanSpec specifies a Huffman encoding.
97     type huffmanSpec struct {
98         // count[i] is the number of codes of length i bits.
99         count [16]byte
100        // value[i] is the decoded value of the i'th codewor
101        value []byte
102    }
103
104     // theHuffmanSpec is the Huffman encoding specifications.
105     // This encoder uses the same Huffman encoding for all image
106     var theHuffmanSpec = [nHuffIndex]huffmanSpec{
107         // Luminance DC.
108         {
109             [16]byte{0, 1, 5, 1, 1, 1, 1, 1, 1, 0, 0, 0,
110             []byte{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
111         },
112         // Luminance AC.
113         {
114             [16]byte{0, 2, 1, 3, 3, 2, 4, 3, 5, 5, 4, 4,
115             []byte{
116                 0x01, 0x02, 0x03, 0x00, 0x04, 0x11,
117                 0x21, 0x31, 0x41, 0x06, 0x13, 0x51,
118                 0x22, 0x71, 0x14, 0x32, 0x81, 0x91,
119                 0x23, 0x42, 0xb1, 0xc1, 0x15, 0x52,
120                 0x24, 0x33, 0x62, 0x72, 0x82, 0x09,
121                 0x17, 0x18, 0x19, 0x1a, 0x25, 0x26,
122                 0x29, 0x2a, 0x34, 0x35, 0x36, 0x37,
123                 0x3a, 0x43, 0x44, 0x45, 0x46, 0x47,
124                 0x4a, 0x53, 0x54, 0x55, 0x56, 0x57,
125                 0x5a, 0x63, 0x64, 0x65, 0x66, 0x67,
126                 0x6a, 0x73, 0x74, 0x75, 0x76, 0x77,
127                 0x7a, 0x83, 0x84, 0x85, 0x86, 0x87,
128                 0x8a, 0x92, 0x93, 0x94, 0x95, 0x96,
129                 0x99, 0x9a, 0xa2, 0xa3, 0xa4, 0xa5,
130                 0xa8, 0xa9, 0xaa, 0xb2, 0xb3, 0xb4,
131                 0xb7, 0xb8, 0xb9, 0xba, 0xc2, 0xc3,
132                 0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xd2,
133                 0xd5, 0xd6, 0xd7, 0xd8, 0xd9, 0xda,
134                 0xe3, 0xe4, 0xe5, 0xe6, 0xe7, 0xe8,
135                 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6,
136                 0xf9, 0xfa,
137             },
138         },
139         // Chrominance DC.
140         {

```

```

141         [16]byte{0, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
142         []byte{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
143     },
144     // Chrominance AC.
145     {
146         [16]byte{0, 2, 1, 2, 4, 4, 3, 4, 7, 5, 4, 4,
147         []byte{
148             0x00, 0x01, 0x02, 0x03, 0x11, 0x04,
149             0x31, 0x06, 0x12, 0x41, 0x51, 0x07,
150             0x13, 0x22, 0x32, 0x81, 0x08, 0x14,
151             0xa1, 0xb1, 0xc1, 0x09, 0x23, 0x33,
152             0x15, 0x62, 0x72, 0xd1, 0x0a, 0x16,
153             0xe1, 0x25, 0xf1, 0x17, 0x18, 0x19,
154             0x27, 0x28, 0x29, 0x2a, 0x35, 0x36,
155             0x39, 0x3a, 0x43, 0x44, 0x45, 0x46,
156             0x49, 0x4a, 0x53, 0x54, 0x55, 0x56,
157             0x59, 0x5a, 0x63, 0x64, 0x65, 0x66,
158             0x69, 0x6a, 0x73, 0x74, 0x75, 0x76,
159             0x79, 0x7a, 0x82, 0x83, 0x84, 0x85,
160             0x88, 0x89, 0x8a, 0x92, 0x93, 0x94,
161             0x97, 0x98, 0x99, 0x9a, 0xa2, 0xa3,
162             0xa6, 0xa7, 0xa8, 0xa9, 0xaa, 0xb2,
163             0xb5, 0xb6, 0xb7, 0xb8, 0xb9, 0xba,
164             0xc4, 0xc5, 0xc6, 0xc7, 0xc8, 0xc9,
165             0xd3, 0xd4, 0xd5, 0xd6, 0xd7, 0xd8,
166             0xe2, 0xe3, 0xe4, 0xe5, 0xe6, 0xe7,
167             0xea, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6,
168             0xf9, 0xfa,
169         },
170     },
171 }
172
173 // huffmanLUT is a compiled look-up table representation of
174 // Each value maps to a uint32 of which the 8 most significa
175 // codeword size in bits and the 24 least significant bits h
176 // The maximum codeword size is 16 bits.
177 type huffmanLUT []uint32
178
179 func (h *huffmanLUT) init(s huffmanSpec) {
180     maxValue := 0
181     for _, v := range s.value {
182         if int(v) > maxValue {
183             maxValue = int(v)
184         }
185     }
186     *h = make([]uint32, maxValue+1)
187     code, k := uint32(0), 0
188     for i := 0; i < len(s.count); i++ {
189         nBits := uint32(i+1) << 24

```

```

190             for j := uint8(0); j < s.count[i]; j++ {
191                 (*h)[s.value[k]] = nBits | code
192                 code++
193                 k++
194             }
195             code <<= 1
196         }
197     }
198
199 // theHuffmanLUT are compiled representations of theHuffmanS
200 var theHuffmanLUT [4]huffmanLUT
201
202 func init() {
203     for i, s := range theHuffmanSpec {
204         theHuffmanLUT[i].init(s)
205     }
206 }
207
208 // writer is a buffered writer.
209 type writer interface {
210     Flush() error
211     Write([]byte) (int, error)
212     WriteByte(byte) error
213 }
214
215 // encoder encodes an image to the JPEG format.
216 type encoder struct {
217     // w is the writer to write to. err is the first err
218     // writing. All attempted writes after the first err
219     w    writer
220     err  error
221     // buf is a scratch buffer.
222     buf  [16]byte
223     // bits and nBits are accumulated bits to write to w
224     bits, nBits uint32
225     // quant is the scaled quantization tables.
226     quant [nQuantIndex][blockSize]byte
227 }
228
229 func (e *encoder) flush() {
230     if e.err != nil {
231         return
232     }
233     e.err = e.w.Flush()
234 }
235
236 func (e *encoder) write(p []byte) {
237     if e.err != nil {
238         return
239     }

```

```

240     _, e.err = e.w.Write(p)
241 }
242
243 func (e *encoder) writeByte(b byte) {
244     if e.err != nil {
245         return
246     }
247     e.err = e.w.WriteByte(b)
248 }
249
250 // emit emits the least significant nBits bits of bits to th
251 // The precondition is bits < 1<<nBits && nBits <= 16.
252 func (e *encoder) emit(bits, nBits uint32) {
253     nBits += e.nBits
254     bits <<= 32 - nBits
255     bits |= e.bits
256     for nBits >= 8 {
257         b := uint8(bits >> 24)
258         e.WriteByte(b)
259         if b == 0xff {
260             e.WriteByte(0x00)
261         }
262         bits <<= 8
263         nBits -= 8
264     }
265     e.bits, e.nBits = bits, nBits
266 }
267
268 // emitHuff emits the given value with the given Huffman enc
269 func (e *encoder) emitHuff(h huffIndex, value int) {
270     x := theHuffmanLUT[h][value]
271     e.emit(x&(1<<24-1), x>>24)
272 }
273
274 // emitHuffRLE emits a run of runLength copies of value enco
275 // Huffman encoder.
276 func (e *encoder) emitHuffRLE(h huffIndex, runLength, value
277     a, b := value, value
278     if a < 0 {
279         a, b = -value, value-1
280     }
281     var nBits uint32
282     if a < 0x100 {
283         nBits = uint32(bitCount[a])
284     } else {
285         nBits = 8 + uint32(bitCount[a>>8])
286     }
287     e.emitHuff(h, runLength<<4|int(nBits))
288     if nBits > 0 {

```

```

289             e.emit(uint32(b)&(1<<nBits-1), nBits)
290         }
291     }
292
293 // writeMarkerHeader writes the header for a marker with the
294 func (e *encoder) writeMarkerHeader(marker uint8, markerlen
295     e.buf[0] = 0xff
296     e.buf[1] = marker
297     e.buf[2] = uint8(markerlen >> 8)
298     e.buf[3] = uint8(markerlen & 0xff)
299     e.write(e.buf[:4])
300 }
301
302 // writeDQT writes the Define Quantization Table marker.
303 func (e *encoder) writeDQT() {
304     markerlen := 2 + int(nQuantIndex)*(1+blockSize)
305     e.writeMarkerHeader(dqtMarker, markerlen)
306     for i := range e.quant {
307         e.WriteByte(uint8(i))
308         e.write(e.quant[i][:])
309     }
310 }
311
312 // writeSOF0 writes the Start Of Frame (Baseline) marker.
313 func (e *encoder) writeSOF0(size image.Point) {
314     markerlen := 8 + 3*nColorComponent
315     e.writeMarkerHeader(sof0Marker, markerlen)
316     e.buf[0] = 8 // 8-bit color.
317     e.buf[1] = uint8(size.Y >> 8)
318     e.buf[2] = uint8(size.Y & 0xff)
319     e.buf[3] = uint8(size.X >> 8)
320     e.buf[4] = uint8(size.X & 0xff)
321     e.buf[5] = nColorComponent
322     for i := 0; i < nColorComponent; i++ {
323         e.buf[3*i+6] = uint8(i + 1)
324         // We use 4:2:0 chroma subsampling.
325         e.buf[3*i+7] = "\x22\x11\x11"[i]
326         e.buf[3*i+8] = "\x00\x01\x01"[i]
327     }
328     e.write(e.buf[:3*(nColorComponent-1)+9])
329 }
330
331 // writeDHT writes the Define Huffman Table marker.
332 func (e *encoder) writeDHT() {
333     markerlen := 2
334     for _, s := range theHuffmanSpec {
335         markerlen += 1 + 16 + len(s.value)
336     }
337     e.writeMarkerHeader(dhtMarker, markerlen)

```

```

338         for i, s := range theHuffmanSpec {
339             e.WriteByte("\x00\x10\x01\x11"[i])
340             e.write(s.count[:])
341             e.write(s.value)
342         }
343     }
344
345     // writeBlock writes a block of pixel data using the given q
346     // returning the post-quantized DC value of the DCT-transfor
347     func (e *encoder) writeBlock(b *block, q quantIndex, prevDC
348         fdct(b)
349         // Emit the DC delta.
350         dc := div(b[0], (8 * int(e.quant[q][0])))
351         e.emitHuffRLE(huffIndex(2*q+0), 0, dc-prevDC)
352         // Emit the AC components.
353         h, runLength := huffIndex(2*q+1), 0
354         for k := 1; k < blockSize; k++ {
355             ac := div(b[unzig[k]], (8 * int(e.quant[q][k]
356             if ac == 0 {
357                 runLength++
358             } else {
359                 for runLength > 15 {
360                     e.emitHuff(h, 0xf0)
361                     runLength -= 16
362                 }
363                 e.emitHuffRLE(h, runLength, ac)
364                 runLength = 0
365             }
366         }
367         if runLength > 0 {
368             e.emitHuff(h, 0x00)
369         }
370         return dc
371     }
372
373     // toYCbCr converts the 8x8 region of m whose top-left corne
374     // YCbCr values.
375     func toYCbCr(m image.Image, p image.Point, yBlock, cbBlock,
376         b := m.Bounds()
377         xmax := b.Max.X - 1
378         ymax := b.Max.Y - 1
379         for j := 0; j < 8; j++ {
380             for i := 0; i < 8; i++ {
381                 r, g, b, _ := m.At(min(p.X+i, xmax),
382                 yy, cb, cr := color.RGBToYCbCr(uint8
383                 yBlock[8*j+i] = int(yy)
384                 cbBlock[8*j+i] = int(cb)
385                 crBlock[8*j+i] = int(cr)
386             }
387         }

```

```

388 }
389
390 // rgbaToYCbCr is a specialized version of toYCbCr for image
391 func rgbaToYCbCr(m *image.RGBA, p image.Point, yBlock, cbBlo
392     b := m.Bounds()
393     xmax := b.Max.X - 1
394     ymax := b.Max.Y - 1
395     for j := 0; j < 8; j++ {
396         sj := p.Y + j
397         if sj > ymax {
398             sj = ymax
399         }
400         offset := (sj-b.Min.Y)*m.Stride - b.Min.X*4
401         for i := 0; i < 8; i++ {
402             sx := p.X + i
403             if sx > xmax {
404                 sx = xmax
405             }
406             pix := m.Pix[offset+sx*4:]
407             yy, cb, cr := color.RGBToYCbCr(pix[0
408             yBlock[8*j+i] = int(yy)
409             cbBlock[8*j+i] = int(cb)
410             crBlock[8*j+i] = int(cr)
411         }
412     }
413 }
414
415 // scale scales the 16x16 region represented by the 4 src bl
416 // dst block.
417 func scale(dst *block, src *[4]block) {
418     for i := 0; i < 4; i++ {
419         dstOff := (i&2)<<4 | (i&1)<<2
420         for y := 0; y < 4; y++ {
421             for x := 0; x < 4; x++ {
422                 j := 16*y + 2*x
423                 sum := src[i][j] + src[i][j+
424                 dst[8*y+x+dstOff] = (sum + 2
425             }
426         }
427     }
428 }
429
430 // sosHeader is the SOS marker "\xff\xda" followed by 12 byt
431 // - the marker length "\x00\x0c",
432 // - the number of components "\x03",
433 // - component 1 uses DC table 0 and AC table 0 "\x01\x
434 // - component 2 uses DC table 1 and AC table 1 "\x02\x
435 // - component 3 uses DC table 1 and AC table 1 "\x03\x
436 // - padding "\x00\x00\x00".

```

```

437 var sosHeader = []byte{
438     0xff, 0xda, 0x00, 0x0c, 0x03, 0x01, 0x00, 0x02,
439     0x11, 0x03, 0x11, 0x00, 0x00, 0x00,
440 }
441
442 // writeSOS writes the StartOfScan marker.
443 func (e *encoder) writeSOS(m image.Image) {
444     e.write(sosHeader)
445     var (
446         // Scratch buffers to hold the YCbCr values.
447         yBlock block
448         cbBlock [4]block
449         crBlock [4]block
450         cBlock block
451         // DC components are delta-encoded.
452         prevDCY, prevDCCb, prevDCCr int
453     )
454     bounds := m.Bounds()
455     rgba, _ := m.(*image.RGBA)
456     for y := bounds.Min.Y; y < bounds.Max.Y; y += 16 {
457         for x := bounds.Min.X; x < bounds.Max.X; x +
458             for i := 0; i < 4; i++ {
459                 xOff := (i & 1) * 8
460                 yOff := (i & 2) * 4
461                 p := image.Pt(x+xOff, y+yOff)
462                 if rgba != nil {
463                     rgbaToYCbCr(rgba, p,
464                 } else {
465                     toYCbCr(m, p, &yBloc
466                 }
467                 prevDCY = e.writeBlock(&yBloc
468             }
469             scale(&cBlock, &cbBlock)
470             prevDCCb = e.writeBlock(&cBlock, 1,
471             scale(&cBlock, &crBlock)
472             prevDCCr = e.writeBlock(&cBlock, 1,
473         }
474     }
475     // Pad the last byte with 1's.
476     e.emit(0x7f, 7)
477 }
478
479 // DefaultQuality is the default quality encoding parameter.
480 const DefaultQuality = 75
481
482 // Options are the encoding parameters.
483 // Quality ranges from 1 to 100 inclusive, higher is better.
484 type Options struct {
485     Quality int

```

```

486 }
487
488 // Encode writes the Image m to w in JPEG 4:2:0 baseline for
489 // options. Default parameters are used if a nil *Options is
490 func Encode(w io.Writer, m image.Image, o *Options) error {
491     b := m.Bounds()
492     if b.Dx() >= 1<<16 || b.Dy() >= 1<<16 {
493         return errors.New("jpeg: image is too large")
494     }
495     var e encoder
496     if ww, ok := w.(writer); ok {
497         e.w = ww
498     } else {
499         e.w = bufio.NewWriter(w)
500     }
501     // Clip quality to [1, 100].
502     quality := DefaultQuality
503     if o != nil {
504         quality = o.Quality
505         if quality < 1 {
506             quality = 1
507         } else if quality > 100 {
508             quality = 100
509         }
510     }
511     // Convert from a quality rating to a scaling factor
512     var scale int
513     if quality < 50 {
514         scale = 5000 / quality
515     } else {
516         scale = 200 - quality*2
517     }
518     // Initialize the quantization tables.
519     for i := range e.quant {
520         for j := range e.quant[i] {
521             x := int(unscaledQuant[i][j])
522             x = (x*scale + 50) / 100
523             if x < 1 {
524                 x = 1
525             } else if x > 255 {
526                 x = 255
527             }
528             e.quant[i][j] = uint8(x)
529         }
530     }
531     // Write the Start Of Image marker.
532     e.buf[0] = 0xff
533     e.buf[1] = 0xd8
534     e.write(e.buf[:2])
535     // Write the quantization tables.

```

```
536         e.writeDQT()
537         // Write the image dimensions.
538         e.writeSOF0(b.Size())
539         // Write the Huffman tables.
540         e.writeDHT()
541         // Write the image data.
542         e.writeSOS(m)
543         // Write the End Of Image marker.
544         e.buf[0] = 0xff
545         e.buf[1] = 0xd9
546         e.write(e.buf[:2])
547         e.flush()
548         return e.err
549     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/image/png/reader.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package png implements a PNG image decoder and encoder.
6 //
7 // The PNG specification is at http://www.w3.org/TR/PNG/.
8 package png
9
10 import (
11     "compress/zlib"
12     "encoding/binary"
13     "fmt"
14     "hash"
15     "hash/crc32"
16     "image"
17     "image/color"
18     "io"
19 )
20
21 // Color type, as per the PNG spec.
22 const (
23     ctGrayscale      = 0
24     ctTrueColor      = 2
25     ctPaletted       = 3
26     ctGrayscaleAlpha = 4
27     ctTrueColorAlpha = 6
28 )
29
30 // A cb is a combination of color type and bit depth.
31 const (
32     cbInvalid = iota
33     cbG1
34     cbG2
35     cbG4
36     cbG8
37     cbGA8
38     cbTC8
39     cbP1
40     cbP2
41     cbP4
```

```

42         cbP8
43         cbTCA8
44         cbG16
45         cbGA16
46         cbTC16
47         cbTCA16
48     )
49
50 // Filter type, as per the PNG spec.
51 const (
52     ftNone    = 0
53     ftSub     = 1
54     ftUp      = 2
55     ftAverage = 3
56     ftPaeth   = 4
57     nFilter   = 5
58 )
59
60 // Decoding stage.
61 // The PNG specification says that the IHDR, PLTE (if present)
62 // chunks must appear in that order. There may be multiple IDAT
63 // chunks must be sequential (i.e. they may not have an IDAT
64 // between them).
65 // http://www.w3.org/TR/PNG/#5ChunkOrdering
66 const (
67     dsStart = iota
68     dsSeenIHDR
69     dsSeenPLTE
70     dsSeenIDAT
71     dsSeenIEND
72 )
73
74 const pngHeader = "\x89PNG\r\n\x1a\n"
75
76 type decoder struct {
77     r          io.Reader
78     img        image.Image
79     crc        hash.Hash32
80     width, height int
81     depth      int
82     palette    color.Palette
83     cb         int
84     stage      int
85     idatLength uint32
86     tmp        [3 * 256]byte
87 }
88
89 // A FormatError reports that the input is not a valid PNG.
90 type FormatError string
91

```

```

92 func (e FormatError) Error() string { return "png: invalid f
93
94 var chunkOrderError = FormatError("chunk out of order")
95
96 // An UnsupportedError reports that the input uses a valid b
97 type UnsupportedError string
98
99 func (e UnsupportedError) Error() string { return "png: unsu
100
101 func abs(x int) int {
102     if x < 0 {
103         return -x
104     }
105     return x
106 }
107
108 func min(a, b int) int {
109     if a < b {
110         return a
111     }
112     return b
113 }
114
115 func (d *decoder) parseIHDR(length uint32) error {
116     if length != 13 {
117         return FormatError("bad IHDR length")
118     }
119     if _, err := io.ReadFull(d.r, d.tmp[:13]); err != ni
120         return err
121     }
122     d.crc.Write(d.tmp[:13])
123     if d.tmp[10] != 0 || d.tmp[11] != 0 || d.tmp[12] !=
124         return UnsupportedError("compression, filter
125     }
126     w := int32(binary.BigEndian.Uint32(d.tmp[0:4]))
127     h := int32(binary.BigEndian.Uint32(d.tmp[4:8]))
128     if w < 0 || h < 0 {
129         return FormatError("negative dimension")
130     }
131     nPixels := int64(w) * int64(h)
132     if nPixels != int64(int(nPixels)) {
133         return UnsupportedError("dimension overflow"
134     }
135     d.cb = cbInvalid
136     d.depth = int(d.tmp[8])
137     switch d.depth {
138     case 1:
139         switch d.tmp[9] {
140         case ctGrayscale:

```

```

141             d.cb = cbG1
142         case ctPaletted:
143             d.cb = cbP1
144     }
145     case 2:
146         switch d.tmp[9] {
147         case ctGrayscale:
148             d.cb = cbG2
149         case ctPaletted:
150             d.cb = cbP2
151         }
152     case 4:
153         switch d.tmp[9] {
154         case ctGrayscale:
155             d.cb = cbG4
156         case ctPaletted:
157             d.cb = cbP4
158         }
159     case 8:
160         switch d.tmp[9] {
161         case ctGrayscale:
162             d.cb = cbG8
163         case ctTrueColor:
164             d.cb = cbTC8
165         case ctPaletted:
166             d.cb = cbP8
167         case ctGrayscaleAlpha:
168             d.cb = cbGA8
169         case ctTrueColorAlpha:
170             d.cb = cbTCA8
171         }
172     case 16:
173         switch d.tmp[9] {
174         case ctGrayscale:
175             d.cb = cbG16
176         case ctTrueColor:
177             d.cb = cbTC16
178         case ctGrayscaleAlpha:
179             d.cb = cbGA16
180         case ctTrueColorAlpha:
181             d.cb = cbTCA16
182         }
183     }
184     if d.cb == cbInvalid {
185         return UnsupportedError(fmt.Sprintf("bit dep
186     })
187     d.width, d.height = int(w), int(h)
188     return d.verifyChecksum()
189 }

```

```

190
191 func (d *decoder) parsePLTE(length uint32) error {
192     np := int(length / 3) // The number of palette entri
193     if length%3 != 0 || np <= 0 || np > 256 || np > 1<<u
194         return FormatError("bad PLTE length")
195     }
196     n, err := io.ReadFull(d.r, d.tmp[:3*np])
197     if err != nil {
198         return err
199     }
200     d.crc.Write(d.tmp[:n])
201     switch d.cb {
202     case cbP1, cbP2, cbP4, cbP8:
203         d.palette = color.Palette(make([]color.Color
204             for i := 0; i < np; i++ {
205                 d.palette[i] = color.RGBA{d.tmp[3*i+
206             }
207     case cbTC8, cbTCA8, cbTC16, cbTCA16:
208         // As per the PNG spec, a PLTE chunk is opti
209         // ignorable) for the ctTrueColor and ctTrue
210     default:
211         return FormatError("PLTE, color type mismatc
212     }
213     return d.verifyChecksum()
214 }
215
216 func (d *decoder) parsetRNS(length uint32) error {
217     if length > 256 {
218         return FormatError("bad tRNS length")
219     }
220     n, err := io.ReadFull(d.r, d.tmp[:length])
221     if err != nil {
222         return err
223     }
224     d.crc.Write(d.tmp[:n])
225     switch d.cb {
226     case cbG8, cbG16:
227         return UnsupportedError("grayscale transpare
228     case cbTC8, cbTC16:
229         return UnsupportedError("truecolor transpare
230     case cbP1, cbP2, cbP4, cbP8:
231         if n > len(d.palette) {
232             return FormatError("bad tRNS length"
233         }
234         for i := 0; i < n; i++ {
235             rgba := d.palette[i].(color.RGBA)
236             d.palette[i] = color.RGBA{rgba.R, rg
237         }
238     case cbGA8, cbGA16, cbTCA8, cbTCA16:
239         return FormatError("tRNS, color type mismatc

```

```

240     }
241     return d.verifyChecksum()
242 }
243
244 // The Paeth filter function, as per the PNG specification.
245 func paeth(a, b, c uint8) uint8 {
246     p := int(a) + int(b) - int(c)
247     pa := abs(p - int(a))
248     pb := abs(p - int(b))
249     pc := abs(p - int(c))
250     if pa <= pb && pa <= pc {
251         return a
252     } else if pb <= pc {
253         return b
254     }
255     return c
256 }
257
258 // Read presents one or more IDAT chunks as one continuous s
259 // intermediate chunk headers and footers). If the PNG data
260 // ... len0 IDAT xxx crc0 len1 IDAT yy crc1 len2 IEND crc2
261 // then this reader presents xxxyy. For well-formed PNG data
262 // immediately before the first Read call is that d.r is pos
263 // first IDAT and xxx, and the decoder state immediately aft
264 // call is that d.r is positioned between yy and crc1.
265 func (d *decoder) Read(p []byte) (int, error) {
266     if len(p) == 0 {
267         return 0, nil
268     }
269     for d.idatLength == 0 {
270         // We have exhausted an IDAT chunk. Verify t
271         if err := d.verifyChecksum(); err != nil {
272             return 0, err
273         }
274         // Read the length and chunk type of the nex
275         // it is an IDAT chunk.
276         if _, err := io.ReadFull(d.r, d.tmp[:8]); er
277             return 0, err
278         }
279         d.idatLength = binary.BigEndian.Uint32(d.tmp
280         if string(d.tmp[4:8]) != "IDAT" {
281             return 0, FormatError("not enough pi
282         }
283         d.crc.Reset()
284         d.crc.Write(d.tmp[4:8])
285     }
286     if int(d.idatLength) < 0 {
287         return 0, UnsupportedError("IDAT chunk lengt
288     }

```

```

289         n, err := d.r.Read(p[:min(len(p), int(d.idatLength))]
290         d.crc.Write(p[:n])
291         d.idatLength -= uint32(n)
292         return n, err
293     }
294
295     // decode decodes the IDAT data into an image.
296     func (d *decoder) decode() (image.Image, error) {
297         r, err := zlib.NewReader(d)
298         if err != nil {
299             return nil, err
300         }
301         defer r.Close()
302         bitsPerPixel := 0
303         maxPalette := uint8(0)
304         var (
305             gray      *image.Gray
306             rgba      *image.RGBA
307             paletted  *image.Paletted
308             nrgba    *image.NRGBA
309             gray16   *image.Gray16
310             rgba64   *image.RGBA64
311             nrgba64 *image.NRGBA64
312             img      image.Image
313         )
314         switch d.cb {
315         case cbG1, cbG2, cbG4, cbG8:
316             bitsPerPixel = d.depth
317             gray = image.NewGray(image.Rect(0, 0, d.wid
318             img = gray
319         case cbGA8:
320             bitsPerPixel = 16
321             nrgba = image.NewNRGBA(image.Rect(0, 0, d.wi
322             img = nrgba
323         case cbTC8:
324             bitsPerPixel = 24
325             rgba = image.NewRGBA(image.Rect(0, 0, d.wid
326             img = rgba
327         case cbP1, cbP2, cbP4, cbP8:
328             bitsPerPixel = d.depth
329             paletted = image.NewPaletted(image.Rect(0, 0
330             img = paletted
331             maxPalette = uint8(len(d.palette) - 1)
332         case cbTCA8:
333             bitsPerPixel = 32
334             nrgba = image.NewNRGBA(image.Rect(0, 0, d.wi
335             img = nrgba
336         case cbG16:
337             bitsPerPixel = 16

```

```

338         gray16 = image.NewGray16(image.Rect(0, 0, d.
339         img = gray16
340     case cbGA16:
341         bitsPerPixel = 32
342         nrgba64 = image.NewNRGBA64(image.Rect(0, 0,
343         img = nrgba64
344     case cbTC16:
345         bitsPerPixel = 48
346         rgba64 = image.NewRGBA64(image.Rect(0, 0, d.
347         img = rgba64
348     case cbTCA16:
349         bitsPerPixel = 64
350         nrgba64 = image.NewNRGBA64(image.Rect(0, 0,
351         img = nrgba64
352     }
353     bytesPerPixel := (bitsPerPixel + 7) / 8
354
355     // cr and pr are the bytes for the current and previ
356     // The +1 is for the per-row filter type, which is a
357     cr := make([]uint8, 1+(bitsPerPixel*d.width+7)/8)
358     pr := make([]uint8, 1+(bitsPerPixel*d.width+7)/8)
359
360     for y := 0; y < d.height; y++ {
361         // Read the decompressed bytes.
362         _, err := io.ReadFull(r, cr)
363         if err != nil {
364             return nil, err
365         }
366
367         // Apply the filter.
368         cdat := cr[1:]
369         pdat := pr[1:]
370         switch cr[0] {
371         case ftNone:
372             // No-op.
373         case ftSub:
374             for i := bytesPerPixel; i < len(cdat)
375             cdat[i] += cdat[i-bytesPerPi
376             }
377         case ftUp:
378             for i := 0; i < len(cdat); i++ {
379                 cdat[i] += pdat[i]
380             }
381         case ftAverage:
382             for i := 0; i < bytesPerPixel; i++ {
383                 cdat[i] += pdat[i] / 2
384             }
385             for i := bytesPerPixel; i < len(cdat)
386             cdat[i] += uint8((int(cdat[i]
387             }

```

```

388     case ftPaeth:
389         for i := 0; i < bytesPerPixel; i++ {
390             cdat[i] += paeth(0, pdat[i],
391                 )
392             for i := bytesPerPixel; i < len(cdat
393                 cdat[i] += paeth(cdat[i-byte
394                 )
395     default:
396         return nil, FormatError("bad filter
397     }
398
399     // Convert from bytes to colors.
400     switch d.cb {
401     case cbG1:
402         for x := 0; x < d.width; x += 8 {
403             b := cdat[x/8]
404             for x2 := 0; x2 < 8 && x+x2
405                 gray.SetGray(x+x2, y
406                 b <<= 1
407             }
408         }
409     case cbG2:
410         for x := 0; x < d.width; x += 4 {
411             b := cdat[x/4]
412             for x2 := 0; x2 < 4 && x+x2
413                 gray.SetGray(x+x2, y
414                 b <<= 2
415             }
416         }
417     case cbG4:
418         for x := 0; x < d.width; x += 2 {
419             b := cdat[x/2]
420             for x2 := 0; x2 < 2 && x+x2
421                 gray.SetGray(x+x2, y
422                 b <<= 4
423             }
424         }
425     case cbG8:
426         for x := 0; x < d.width; x++ {
427             gray.SetGray(x, y, color.Gra
428         }
429     case cbGA8:
430         for x := 0; x < d.width; x++ {
431             ycol := cdat[2*x+0]
432             nrgba.SetNRGBA(x, y, color.N
433         }
434     case cbTC8:
435         for x := 0; x < d.width; x++ {
436             rgba.SetRGBA(x, y, color.RGB

```

```

437     }
438 case cbP1:
439     for x := 0; x < d.width; x += 8 {
440         b := cdat[x/8]
441         for x2 := 0; x2 < 8 && x+x2
442             idx := b >> 7
443             if idx > maxPalette
444                 return nil,
445             }
446         paletted.SetColorInd
447             b <<= 1
448     }
449     }
450 case cbP2:
451     for x := 0; x < d.width; x += 4 {
452         b := cdat[x/4]
453         for x2 := 0; x2 < 4 && x+x2
454             idx := b >> 6
455             if idx > maxPalette
456                 return nil,
457             }
458         paletted.SetColorInd
459             b <<= 2
460     }
461     }
462 case cbP4:
463     for x := 0; x < d.width; x += 2 {
464         b := cdat[x/2]
465         for x2 := 0; x2 < 2 && x+x2
466             idx := b >> 4
467             if idx > maxPalette
468                 return nil,
469             }
470         paletted.SetColorInd
471             b <<= 4
472     }
473     }
474 case cbP8:
475     for x := 0; x < d.width; x++ {
476         if cdat[x] > maxPalette {
477             return nil, FormatEr
478         }
479         paletted.SetColorIndex(x, y,
480     }
481 case cbTCA8:
482     for x := 0; x < d.width; x++ {
483         nrgba.SetNRGBA(x, y, color.N
484     }
485 case cbG16:

```

```

486         for x := 0; x < d.width; x++ {
487             ycol := uint16(cdat[2*x+0])<
488                 gray16.SetGray16(x, y, color
489         }
490     case cbGA16:
491         for x := 0; x < d.width; x++ {
492             ycol := uint16(cdat[4*x+0])<
493             acol := uint16(cdat[4*x+2])<
494             nrgba64.SetNRGBA64(x, y, col
495         }
496     case cbTC16:
497         for x := 0; x < d.width; x++ {
498             rcol := uint16(cdat[6*x+0])<
499             gcol := uint16(cdat[6*x+2])<
500             bcol := uint16(cdat[6*x+4])<
501             rgba64.SetRGBA64(x, y, color
502         }
503     case cbTCA16:
504         for x := 0; x < d.width; x++ {
505             rcol := uint16(cdat[8*x+0])<
506             gcol := uint16(cdat[8*x+2])<
507             bcol := uint16(cdat[8*x+4])<
508             acol := uint16(cdat[8*x+6])<
509             nrgba64.SetNRGBA64(x, y, col
510         }
511     }
512
513     // The current row for y is the previous row
514     pr, cr = cr, pr
515 }
516
517 // Check for EOF, to verify the zlib checksum.
518 n, err := r.Read(pr[:1])
519 if err != io.EOF {
520     return nil, FormatError(err.Error())
521 }
522 if n != 0 || d.idatLength != 0 {
523     return nil, FormatError("too much pixel data
524 }
525
526 return img, nil
527 }
528
529 func (d *decoder) parseIDAT(length uint32) (err error) {
530     d.idatLength = length
531     d.img, err = d.decode()
532     if err != nil {
533         return err
534     }
535     return d.verifyChecksum()

```

```

536 }
537
538 func (d *decoder) parseIEND(length uint32) error {
539     if length != 0 {
540         return FormatError("bad IEND length")
541     }
542     return d.verifyChecksum()
543 }
544
545 func (d *decoder) parseChunk() error {
546     // Read the length and chunk type.
547     n, err := io.ReadFull(d.r, d.tmp[:8])
548     if err != nil {
549         return err
550     }
551     length := binary.BigEndian.Uint32(d.tmp[:4])
552     d.crc.Reset()
553     d.crc.Write(d.tmp[4:8])
554
555     // Read the chunk data.
556     switch string(d.tmp[4:8]) {
557     case "IHDR":
558         if d.stage != dsStart {
559             return chunkOrderError
560         }
561         d.stage = dsSeenIHDR
562         return d.parseIHDR(length)
563     case "PLTE":
564         if d.stage != dsSeenIHDR {
565             return chunkOrderError
566         }
567         d.stage = dsSeenPLTE
568         return d.parsePLTE(length)
569     case "tRNS":
570         if d.stage != dsSeenPLTE {
571             return chunkOrderError
572         }
573         return d.parseTRNS(length)
574     case "IDAT":
575         if d.stage < dsSeenIHDR || d.stage > dsSeenI
576             return chunkOrderError
577         }
578         d.stage = dsSeenIDAT
579         return d.parseIDAT(length)
580     case "IEND":
581         if d.stage != dsSeenIDAT {
582             return chunkOrderError
583         }
584         d.stage = dsSeenIEND

```

```

585         return d.parseIEND(length)
586     }
587     // Ignore this chunk (of a known length).
588     var ignored [4096]byte
589     for length > 0 {
590         n, err = io.ReadFull(d.r, ignored[:min(len(i
591             if err != nil {
592                 return err
593             }
594             d.crc.Write(ignored[:n])
595             length -= uint32(n)
596         }
597     return d.verifyChecksum()
598 }
599
600 func (d *decoder) verifyChecksum() error {
601     if _, err := io.ReadFull(d.r, d.tmp[:4]); err != nil
602         return err
603     }
604     if binary.BigEndian.Uint32(d.tmp[:4]) != d.crc.Sum32
605         return FormatError("invalid checksum")
606     }
607     return nil
608 }
609
610 func (d *decoder) checkHeader() error {
611     _, err := io.ReadFull(d.r, d.tmp[:len(pngHeader)])
612     if err != nil {
613         return err
614     }
615     if string(d.tmp[:len(pngHeader)]) != pngHeader {
616         return FormatError("not a PNG file")
617     }
618     return nil
619 }
620
621 // Decode reads a PNG image from r and returns it as an imag
622 // The type of Image returned depends on the PNG contents.
623 func Decode(r io.Reader) (image.Image, error) {
624     d := &decoder{
625         r:    r,
626         crc:  crc32.NewIEEE(),
627     }
628     if err := d.checkHeader(); err != nil {
629         if err == io.EOF {
630             err = io.ErrUnexpectedEOF
631         }
632         return nil, err
633     }

```

```

634         for d.stage != dsSeenIEND {
635             if err := d.parseChunk(); err != nil {
636                 if err == io.EOF {
637                     err = io.ErrUnexpectedEOF
638                 }
639                 return nil, err
640             }
641         }
642         return d.img, nil
643     }
644
645     // DecodeConfig returns the color model and dimensions of a
646     // decoding the entire image.
647     func DecodeConfig(r io.Reader) (image.Config, error) {
648         d := &decoder{
649             r:    r,
650             crc:  crc32.NewIEEE(),
651         }
652         if err := d.checkHeader(); err != nil {
653             if err == io.EOF {
654                 err = io.ErrUnexpectedEOF
655             }
656             return image.Config{}, err
657         }
658         for {
659             if err := d.parseChunk(); err != nil {
660                 if err == io.EOF {
661                     err = io.ErrUnexpectedEOF
662                 }
663                 return image.Config{}, err
664             }
665             if d.stage == dsSeenIHDR && d.cb != cbP8 {
666                 break
667             }
668             if d.stage == dsSeenPLTE && d.cb == cbP8 {
669                 break
670             }
671         }
672         var cm color.Model
673         switch d.cb {
674         case cbG1, cbG2, cbG4, cbG8:
675             cm = color.GrayModel
676         case cbGA8:
677             cm = color.NRGBAModel
678         case cbTC8:
679             cm = color.RGBAModel
680         case cbP1, cbP2, cbP4, cbP8:
681             cm = d.palette
682         case cbTCA8:
683             cm = color.NRGBAModel

```

```
684         case cbG16:
685             cm = color.Gray16Model
686         case cbGA16:
687             cm = color.NRGBA64Model
688         case cbTC16:
689             cm = color.RGBA64Model
690         case cbTCA16:
691             cm = color.NRGBA64Model
692     }
693     return image.Config{
694         ColorModel: cm,
695         Width:      d.width,
696         Height:     d.height,
697     }, nil
698 }
699
700 func init() {
701     image.RegisterFormat("png", pngHeader, Decode, Decod
702 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/image/png/writer.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package png
6
7 import (
8     "bufio"
9     "compress/zlib"
10    "hash/crc32"
11    "image"
12    "image/color"
13    "io"
14    "strconv"
15 )
16
17 type encoder struct {
18     w      io.Writer
19     m      image.Image
20     cb     int
21     err    error
22     header [8]byte
23     footer [4]byte
24     tmp    [3 * 256]byte
25 }
26
27 // Big-endian.
28 func writeUint32(b []uint8, u uint32) {
29     b[0] = uint8(u >> 24)
30     b[1] = uint8(u >> 16)
31     b[2] = uint8(u >> 8)
32     b[3] = uint8(u >> 0)
33 }
34
35 type opaquer interface {
36     Opaque() bool
37 }
38
39 // Returns whether or not the image is fully opaque.
40 func opaque(m image.Image) bool {
41     if o, ok := m.(opaquer); ok {
```

```

42         return o.Opaque()
43     }
44     b := m.Bounds()
45     for y := b.Min.Y; y < b.Max.Y; y++ {
46         for x := b.Min.X; x < b.Max.X; x++ {
47             _, _, _, a := m.At(x, y).RGBA()
48             if a != 0xffff {
49                 return false
50             }
51         }
52     }
53     return true
54 }
55
56 // The absolute value of a byte interpreted as a signed int8
57 func abs8(d uint8) int {
58     if d < 128 {
59         return int(d)
60     }
61     return 256 - int(d)
62 }
63
64 func (e *encoder) writeChunk(b []byte, name string) {
65     if e.err != nil {
66         return
67     }
68     n := uint32(len(b))
69     if int(n) != len(b) {
70         e.err = UnsupportedError(name + " chunk is t
71         return
72     }
73     writeUint32(e.header[0:4], n)
74     e.header[4] = name[0]
75     e.header[5] = name[1]
76     e.header[6] = name[2]
77     e.header[7] = name[3]
78     crc := crc32.NewIEEE()
79     crc.Write(e.header[4:8])
80     crc.Write(b)
81     writeUint32(e.footer[0:4], crc.Sum32())
82
83     _, e.err = e.w.Write(e.header[0:8])
84     if e.err != nil {
85         return
86     }
87     _, e.err = e.w.Write(b)
88     if e.err != nil {
89         return
90     }
91     _, e.err = e.w.Write(e.footer[0:4])

```

```

92 }
93
94 func (e *encoder) writeIHDR() {
95     b := e.m.Bounds()
96     writeUint32(e.tmp[0:4], uint32(b.Dx()))
97     writeUint32(e.tmp[4:8], uint32(b.Dy()))
98     // Set bit depth and color type.
99     switch e.cb {
100    case cbG8:
101        e.tmp[8] = 8
102        e.tmp[9] = ctGrayscale
103    case cbTC8:
104        e.tmp[8] = 8
105        e.tmp[9] = ctTrueColor
106    case cbP8:
107        e.tmp[8] = 8
108        e.tmp[9] = ctPaletted
109    case cbTCA8:
110        e.tmp[8] = 8
111        e.tmp[9] = ctTrueColorAlpha
112    case cbG16:
113        e.tmp[8] = 16
114        e.tmp[9] = ctGrayscale
115    case cbTC16:
116        e.tmp[8] = 16
117        e.tmp[9] = ctTrueColor
118    case cbTCA16:
119        e.tmp[8] = 16
120        e.tmp[9] = ctTrueColorAlpha
121    }
122    e.tmp[10] = 0 // default compression method
123    e.tmp[11] = 0 // default filter method
124    e.tmp[12] = 0 // non-interlaced
125    e.writeChunk(e.tmp[0:13], "IHDR")
126 }
127
128 func (e *encoder) writePLTE(p color.Palette) {
129     if len(p) < 1 || len(p) > 256 {
130         e.err = FormatError("bad palette length: " +
131             return
132     }
133     for i, c := range p {
134         r, g, b, _ := c.RGBA()
135         e.tmp[3*i+0] = uint8(r >> 8)
136         e.tmp[3*i+1] = uint8(g >> 8)
137         e.tmp[3*i+2] = uint8(b >> 8)
138     }
139     e.writeChunk(e.tmp[0:3*len(p)], "PLTE")
140 }

```

```

141
142 func (e *encoder) maybewritetRNS(p color.Palette) {
143     last := -1
144     for i, c := range p {
145         _, _, _, a := c.RGBA()
146         if a != 0xffff {
147             last = i
148         }
149         e.tmp[i] = uint8(a >> 8)
150     }
151     if last == -1 {
152         return
153     }
154     e.writeChunk(e.tmp[:last+1], "tRNS")
155 }
156
157 // An encoder is an io.Writer that satisfies writes by writi
158 // including an 8-byte header and 4-byte CRC checksum per Wr
159 // should be relatively infrequent, since writeIDATs uses a
160 //
161 // This method should only be called from writeIDATs (via wr
162 // No other code should treat an encoder as an io.Writer.
163 func (e *encoder) Write(b []byte) (int, error) {
164     e.writeChunk(b, "IDAT")
165     if e.err != nil {
166         return 0, e.err
167     }
168     return len(b), nil
169 }
170
171 // Chooses the filter to use for encoding the current row, a
172 // The return value is the index of the filter and also of t
173 func filter(cr *[nFilter][]byte, pr []byte, bpp int) int {
174     // We try all five filter types, and pick the one th
175     // This is the same heuristic that libpng uses, alth
176     // estimated most likely to be minimal (ftUp, ftPaet
177     // in their enumeration order (ftNone, ftSub, ftUp,
178     cdat0 := cr[0][1:]
179     cdat1 := cr[1][1:]
180     cdat2 := cr[2][1:]
181     cdat3 := cr[3][1:]
182     cdat4 := cr[4][1:]
183     pdat := pr[1:]
184     n := len(cdat0)
185
186     // The up filter.
187     sum := 0
188     for i := 0; i < n; i++ {
189         cdat2[i] = cdat0[i] - pdat[i]

```

```

190         sum += abs8(cdat2[i])
191     }
192     best := sum
193     filter := ftUp
194
195     // The Paeth filter.
196     sum = 0
197     for i := 0; i < bpp; i++ {
198         cdat4[i] = cdat0[i] - paeth(0, pdat[i], 0)
199         sum += abs8(cdat4[i])
200     }
201     for i := bpp; i < n; i++ {
202         cdat4[i] = cdat0[i] - paeth(cdat0[i-bpp], pd
203         sum += abs8(cdat4[i])
204         if sum >= best {
205             break
206         }
207     }
208     if sum < best {
209         best = sum
210         filter = ftPaeth
211     }
212
213     // The none filter.
214     sum = 0
215     for i := 0; i < n; i++ {
216         sum += abs8(cdat0[i])
217         if sum >= best {
218             break
219         }
220     }
221     if sum < best {
222         best = sum
223         filter = ftNone
224     }
225
226     // The sub filter.
227     sum = 0
228     for i := 0; i < bpp; i++ {
229         cdat1[i] = cdat0[i]
230         sum += abs8(cdat1[i])
231     }
232     for i := bpp; i < n; i++ {
233         cdat1[i] = cdat0[i] - cdat0[i-bpp]
234         sum += abs8(cdat1[i])
235         if sum >= best {
236             break
237         }
238     }
239     if sum < best {

```

```

240         best = sum
241         filter = ftSub
242     }
243
244     // The average filter.
245     sum = 0
246     for i := 0; i < bpp; i++ {
247         cdat3[i] = cdat0[i] - pdat[i]/2
248         sum += abs8(cdat3[i])
249     }
250     for i := bpp; i < n; i++ {
251         cdat3[i] = cdat0[i] - uint8((int(cdat0[i-bpp
252         sum += abs8(cdat3[i])
253         if sum >= best {
254             break
255         }
256     }
257     if sum < best {
258         best = sum
259         filter = ftAverage
260     }
261
262     return filter
263 }
264
265 func writeImage(w io.Writer, m image.Image, cb int) error {
266     zw := zlib.NewWriter(w)
267     defer zw.Close()
268
269     bpp := 0 // Bytes per pixel.
270
271     switch cb {
272     case cbG8:
273         bpp = 1
274     case cbTC8:
275         bpp = 3
276     case cbP8:
277         bpp = 1
278     case cbTCA8:
279         bpp = 4
280     case cbTC16:
281         bpp = 6
282     case cbTCA16:
283         bpp = 8
284     case cbG16:
285         bpp = 2
286     }
287     // cr[*] and pr are the bytes for the current and pr
288     // cr[0] is unfiltered (or equivalently, filtered wi

```

```

289 // cr[ft], for non-zero filter types ft, are buffers
290 // other PNG filter types. These buffers are allocat
291 // The +1 is for the per-row filter type, which is a
292 b := m.Bounds()
293 var cr [nFilter][]uint8
294 for i := range cr {
295     cr[i] = make([]uint8, 1+bpp*b.Dx())
296     cr[i][0] = uint8(i)
297 }
298 pr := make([]uint8, 1+bpp*b.Dx())
299
300 for y := b.Min.Y; y < b.Max.Y; y++ {
301     // Convert from colors to bytes.
302     i := 1
303     switch cb {
304     case cbG8:
305         for x := b.Min.X; x < b.Max.X; x++ {
306             c := color.GrayModel.Convert
307             cr[0][i] = c.Y
308             i++
309         }
310     case cbTC8:
311         // We have previously verified that
312         cr0 := cr[0]
313         if rgba, _ := m.(*image.RGBA); rgba
314             j0 := (y - b.Min.Y) * rgba.S
315             j1 := j0 + b.Dx()*4
316             for j := j0; j < j1; j += 4
317                 cr0[i+0] = rgba.Pix[
318                 cr0[i+1] = rgba.Pix[
319                 cr0[i+2] = rgba.Pix[
320                 i += 3
321             }
322         } else {
323             for x := b.Min.X; x < b.Max.
324                 r, g, b, _ := m.At(x
325                 cr0[i+0] = uint8(r >
326                 cr0[i+1] = uint8(g >
327                 cr0[i+2] = uint8(b >
328                 i += 3
329             }
330         }
331     case cbP8:
332         if p, _ := m.(*image.Paletted); p !=
333             offset := (y - b.Min.Y) * p.
334             copy(cr[0][1:], p.Pix[offset
335         } else {
336             pi := m.(image.PalettedImage
337             for x := b.Min.X; x < b.Max.

```

```

338                                     cr[0][i] = pi.ColorI
339                                     i += 1
340                                 }
341                             }
342     case cbTCA8:
343         // Convert from image.Image (which i
344         for x := b.Min.X; x < b.Max.X; x++ {
345             c := color.NRGBAModel.Conver
346             cr[0][i+0] = c.R
347             cr[0][i+1] = c.G
348             cr[0][i+2] = c.B
349             cr[0][i+3] = c.A
350             i += 4
351         }
352     case cbG16:
353         for x := b.Min.X; x < b.Max.X; x++ {
354             c := color.Gray16Model.Conve
355             cr[0][i+0] = uint8(c.Y >> 8)
356             cr[0][i+1] = uint8(c.Y)
357             i += 2
358         }
359     case cbTC16:
360         // We have previously verified that
361         for x := b.Min.X; x < b.Max.X; x++ {
362             r, g, b, _ := m.At(x, y).RGBE
363             cr[0][i+0] = uint8(r >> 8)
364             cr[0][i+1] = uint8(r)
365             cr[0][i+2] = uint8(g >> 8)
366             cr[0][i+3] = uint8(g)
367             cr[0][i+4] = uint8(b >> 8)
368             cr[0][i+5] = uint8(b)
369             i += 6
370         }
371     case cbTCA16:
372         // Convert from image.Image (which i
373         for x := b.Min.X; x < b.Max.X; x++ {
374             c := color.NRGBA64Model.Conv
375             cr[0][i+0] = uint8(c.R >> 8)
376             cr[0][i+1] = uint8(c.R)
377             cr[0][i+2] = uint8(c.G >> 8)
378             cr[0][i+3] = uint8(c.G)
379             cr[0][i+4] = uint8(c.B >> 8)
380             cr[0][i+5] = uint8(c.B)
381             cr[0][i+6] = uint8(c.A >> 8)
382             cr[0][i+7] = uint8(c.A)
383             i += 8
384         }
385     }
386
387     // Apply the filter.

```

```

388         f := filter(&cr, pr, bpp)
389
390         // Write the compressed bytes.
391         if _, err := zw.Write(cr[f]); err != nil {
392             return err
393         }
394
395         // The current row for y is the previous row
396         pr, cr[0] = cr[0], pr
397     }
398     return nil
399 }
400
401 // Write the actual image data to one or more IDAT chunks.
402 func (e *encoder) writeIDATs() {
403     if e.err != nil {
404         return
405     }
406     var bw *bufio.Writer
407     bw = bufio.NewWriterSize(e, 1<<15)
408     e.err = writeImage(bw, e.m, e.cb)
409     if e.err != nil {
410         return
411     }
412     e.err = bw.Flush()
413 }
414
415 func (e *encoder) writeIEND() { e.writeChunk(e.tmp[0:0], "IE
416
417 // Encode writes the Image m to w in PNG format. Any Image m
418 // images that are not image.NRGBA might be encoded lossily.
419 func Encode(w io.Writer, m image.Image) error {
420     // Obviously, negative widths and heights are invalid
421     // spec section 11.2.2 says that zero is invalid. Ex
422     // also rejected.
423     mw, mh := int64(m.Bounds().Dx()), int64(m.Bounds().D
424     if mw <= 0 || mh <= 0 || mw >= 1<<32 || mh >= 1<<32
425         return FormatError("invalid image size: " +
426     }
427
428     var e encoder
429     e.w = w
430     e.m = m
431
432     var pal color.Palette
433     // cbP8 encoding needs PalettedImage's ColorIndexAt
434     if _, ok := m.(image.PalettedImage); ok {
435         pal, _ = m.ColorModel().(color.Palette)
436     }

```

```

437         if pal != nil {
438             e.cb = cbP8
439         } else {
440             switch m.ColorModel() {
441             case color.GrayModel:
442                 e.cb = cbG8
443             case color.Gray16Model:
444                 e.cb = cbG16
445             case color.RGBAModel, color.NRGBAModel, color.CMYKModel:
446                 if opaque(m) {
447                     e.cb = cbTC8
448                 } else {
449                     e.cb = cbTCA8
450                 }
451             default:
452                 if opaque(m) {
453                     e.cb = cbTC16
454                 } else {
455                     e.cb = cbTCA16
456                 }
457             }
458         }
459
460         _, e.err = io.WriteString(w, pngHeader)
461         e.writeIHDR()
462         if pal != nil {
463             e.writePLTE(pal)
464             e.maybeWritetRNS(pal)
465         }
466         e.writeIDATs()
467         e.writeIEND()
468         return e.err
469     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/index/suffixarray/qsufsort.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This algorithm is based on "Faster Suffix Sorting"
6 //   by N. Jesper Larsson and Kunihiko Sadakane
7 // paper: http://www.larsson.dogma.net/ssrev-tr.pdf
8 // code:  http://www.larsson.dogma.net/qsufsort.c
9
10 // This algorithm computes the suffix array sa by computing
11 // Consecutive groups of suffixes in sa are labeled as sorted
12 // unsorted groups. For a given pass of the sorter, all suffixes
13 // up to their first h characters, and sa is h-ordered. Suffixes
14 // final positions and unambiguously sorted in h-order are i
15 // Consecutive groups of suffixes with identical first h characters
16 // unsorted group. In each pass of the algorithm, unsorted groups
17 // according to the group number of their following suffix.
18
19 // In the implementation, if sa[i] is negative, it indicates
20 // the first element of a sorted group of length -sa[i], and
21 // An unsorted group sa[i:k] is given the group number of the
22 // last element, k-1. The group numbers are stored in the inverse
23 // and when all groups are sorted, this slice is the inverse
24
25 package suffixarray
26
27 import "sort"
28
29 func qsufsort(data []byte) []int {
30     // initial sorting by first byte of suffix
31     sa := sortedByFirstByte(data)
32     if len(sa) < 2 {
33         return sa
34     }
35     // initialize the group lookup table
36     // this becomes the inverse of the suffix array when
37     inv := initGroups(sa, data)
38
39     // the index starts 1-ordered
40     sufSortable := &suffixSortable{sa: sa, inv: inv, h:
41
```

```

42     for sa[0] > -len(sa) { // until all suffixes are one
43         // The suffixes are h-ordered, make them 2*h
44         pi := 0 // pi is first position of first group
45         sl := 0 // sl is negated length of sorted group
46         for pi < len(sa) {
47             if s := sa[pi]; s < 0 { // if pi starts sorted group
48                 pi -= s // skip over sorted group
49                 sl += s // add negated length
50             } else { // if pi starts unsorted group
51                 if sl != 0 {
52                     sa[pi+sl] = sl // combine sorted group
53                     sl = 0
54                 }
55                 pk := inv[s] + 1 // pk-1 is next unsorted group
56                 sufSortable.sa = sa[pi:pk]
57                 sort.Sort(sufSortable)
58                 sufSortable.updateGroups(pi)
59                 pi = pk // next group
60             }
61         }
62         if sl != 0 { // if the array ends with a sorted group
63             sa[pi+sl] = sl // combine sorted group
64         }
65
66         sufSortable.h *= 2 // double sorted depth
67     }
68
69     for i := range sa { // reconstruct suffix array from
70         sa[inv[i]] = i
71     }
72     return sa
73 }
74
75 func sortedByFirstByte(data []byte) []int {
76     // total byte counts
77     var count [256]int
78     for _, b := range data {
79         count[b]++
80     }
81     // make count[b] equal index of first occurrence of
82     sum := 0
83     for b := range count {
84         count[b], sum = sum, count[b]+sum
85     }
86     // iterate through bytes, placing index into the correct
87     sa := make([]int, len(data))
88     for i, b := range data {
89         sa[count[b]] = i
90         count[b]++
91     }

```

```

92         return sa
93     }
94
95     func initGroups(sa []int, data []byte) []int {
96         // label contiguous same-letter groups with the same
97         inv := make([]int, len(data))
98         prevGroup := len(sa) - 1
99         groupByte := data[sa[prevGroup]]
100        for i := len(sa) - 1; i >= 0; i-- {
101            if b := data[sa[i]]; b < groupByte {
102                if prevGroup == i+1 {
103                    sa[i+1] = -1
104                }
105                groupByte = b
106                prevGroup = i
107            }
108            inv[sa[i]] = prevGroup
109            if prevGroup == 0 {
110                sa[0] = -1
111            }
112        }
113        // Separate out the final suffix to the start of its
114        // This is necessary to ensure the suffix "a" is bef
115        // when using a potentially unstable sort.
116        lastByte := data[len(data)-1]
117        s := -1
118        for i := range sa {
119            if sa[i] >= 0 {
120                if data[sa[i]] == lastByte && s == -
121                    s = i
122            }
123            if sa[i] == len(sa)-1 {
124                sa[i], sa[s] = sa[s], sa[i]
125                inv[sa[s]] = s
126                sa[s] = -1 // mark it as an
127                break
128            }
129        }
130    }
131    return inv
132 }
133
134 type suffixSortable struct {
135     sa []int
136     inv []int
137     h int
138     buf []int // common scratch space
139 }
140

```

```

141 func (x *suffixSortable) Len() int           { return len(x.
142 func (x *suffixSortable) Less(i, j int) bool { return x.inv[
143 func (x *suffixSortable) Swap(i, j int)      { x.sa[i], x.sa
144
145 func (x *suffixSortable) updateGroups(offset int) {
146     bounds := x.buf[0:0]
147     group := x.inv[x.sa[0]+x.h]
148     for i := 1; i < len(x.sa); i++ {
149         if g := x.inv[x.sa[i]+x.h]; g > group {
150             bounds = append(bounds, i)
151             group = g
152         }
153     }
154     bounds = append(bounds, len(x.sa))
155     x.buf = bounds
156
157     // update the group numberings after all new groups
158     prev := 0
159     for _, b := range bounds {
160         for i := prev; i < b; i++ {
161             x.inv[x.sa[i]] = offset + b - 1
162         }
163         if b-prev == 1 {
164             x.sa[prev] = -1
165         }
166         prev = b
167     }
168 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/index/suffixarray/suffixarray.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package suffixarray implements substring search in logarithmic
6 // an in-memory suffix array.
7 //
8 // Example use:
9 //
10 //     // create index for some data
11 //     index := suffixarray.New(data)
12 //
13 //     // lookup byte slice s
14 //     offsets1 := index.Lookup(s, -1) // the list of all i
15 //     offsets2 := index.Lookup(s, 3) // the list of at mo
16 //
17 package suffixarray
18
19 import (
20     "bytes"
21     "encoding/binary"
22     "io"
23     "regexp"
24     "sort"
25 )
26
27 // Index implements a suffix array for fast substring search
28 type Index struct {
29     data []byte
30     sa   []int // suffix array for data; len(sa) == len(data)
31 }
32
33 // New creates a new Index for data.
34 // Index creation time is O(N*log(N)) for N = len(data).
35 func New(data []byte) *Index {
36     return &Index{data, qsufsort(data)}
37 }
38
39 // writeInt writes an int x to w using buf to buffer the write
40 func writeInt(w io.Writer, buf []byte, x int) error {
41     binary.PutVarint(buf, int64(x))
42 }
```

```

42         _, err := w.Write(buf[0:binary.MaxVarintLen64])
43         return err
44     }
45
46     // readInt reads an int x from r using buf to buffer the read
47     func readInt(r io.Reader, buf []byte) (int, error) {
48         _, err := io.ReadFull(r, buf[0:binary.MaxVarintLen64])
49         x, _ := binary.Varint(buf)
50         return int(x), err
51     }
52
53     // writeSlice writes data[:n] to w and returns n.
54     // It uses buf to buffer the write.
55     func writeSlice(w io.Writer, buf []byte, data []int) (n int,
56         // encode as many elements as fit into buf
57         p := binary.MaxVarintLen64
58         for ; n < len(data) && p+binary.MaxVarintLen64 <= len(buf); n++ {
59             p += binary.PutUvarint(buf[p:], uint64(data[n]))
60         }
61
62         // update buffer size
63         binary.PutVarint(buf, int64(p))
64
65         // write buffer
66         _, err = w.Write(buf[0:p])
67         return n, err
68     }
69
70     // readSlice reads data[:n] from r and returns n.
71     // It uses buf to buffer the read.
72     func readSlice(r io.Reader, buf []byte, data []int) (n int,
73         // read buffer size
74         var size int
75         size, err = readInt(r, buf)
76         if err != nil {
77             return 0, err
78         }
79
80         // read buffer w/o the size
81         if _, err = io.ReadFull(r, buf[binary.MaxVarintLen64:]); err != nil {
82             return 0, err
83         }
84
85         // decode as many elements as present in buf
86         for p := binary.MaxVarintLen64; p < size; n++ {
87             x, w := binary.Uvarint(buf[p:])
88             data[n] = int(x)
89             p += w
90         }
91

```

```

92         return
93     }
94
95     const bufSize = 16 << 10 // reasonable for BenchmarkSaveRest
96
97     // Read reads the index from r into x; x must not be nil.
98     func (x *Index) Read(r io.Reader) error {
99         // buffer for all reads
100        buf := make([]byte, bufSize)
101
102        // read length
103        n, err := readInt(r, buf)
104        if err != nil {
105            return err
106        }
107
108        // allocate space
109        if 2*n < cap(x.data) || cap(x.data) < n {
110            // new data is significantly smaller or larg
111            // existing buffers - allocate new ones
112            x.data = make([]byte, n)
113            x.sa = make([]int, n)
114        } else {
115            // re-use existing buffers
116            x.data = x.data[0:n]
117            x.sa = x.sa[0:n]
118        }
119
120        // read data
121        if _, err := io.ReadFull(r, x.data); err != nil {
122            return err
123        }
124
125        // read index
126        for sa := x.sa; len(sa) > 0; {
127            n, err := readSlice(r, buf, sa)
128            if err != nil {
129                return err
130            }
131            sa = sa[n:]
132        }
133        return nil
134    }
135
136    // Write writes the index x to w.
137    func (x *Index) Write(w io.Writer) error {
138        // buffer for all writes
139        buf := make([]byte, bufSize)
140

```

```

141         // write length
142         if err := writeInt(w, buf, len(x.data)); err != nil
143             return err
144     }
145
146     // write data
147     if _, err := w.Write(x.data); err != nil {
148         return err
149     }
150
151     // write index
152     for sa := x.sa; len(sa) > 0; {
153         n, err := writeSlice(w, buf, sa)
154         if err != nil {
155             return err
156         }
157         sa = sa[n:]
158     }
159     return nil
160 }
161
162 // Bytes returns the data over which the index was created.
163 // It must not be modified.
164 //
165 func (x *Index) Bytes() []byte {
166     return x.data
167 }
168
169 func (x *Index) at(i int) []byte {
170     return x.data[x.sa[i]:]
171 }
172
173 // lookupAll returns a slice into the matching region of the
174 // The runtime is O(log(N)*len(s)).
175 func (x *Index) lookupAll(s []byte) []int {
176     // find matching suffix index range [i:j]
177     // find the first index where s would be the prefix
178     i := sort.Search(len(x.sa), func(i int) bool { retur
179     // starting at i, find the first index at which s is
180     j := i + sort.Search(len(x.sa)-i, func(j int) bool {
181     return x.sa[i:j]
182 }
183
184 // Lookup returns an unsorted list of at most n indices wher
185 // occurs in the indexed data. If n < 0, all occurrences are
186 // The result is nil if s is empty, s is not found, or n ==
187 // Lookup time is O(log(N)*len(s) + len(result)) where N is
188 // size of the indexed data.
189 //

```

```

190 func (x *Index) Lookup(s []byte, n int) (result []int) {
191     if len(s) > 0 && n != 0 {
192         matches := x.LookupAll(s)
193         if n < 0 || len(matches) < n {
194             n = len(matches)
195         }
196         // 0 <= n <= len(matches)
197         if n > 0 {
198             result = make([]int, n)
199             copy(result, matches)
200         }
201     }
202     return
203 }
204
205 // FindAllIndex returns a sorted list of non-overlapping mat
206 // regular expression r, where a match is a pair of indices
207 // the matched slice of x.Bytes(). If n < 0, all matches are
208 // in successive order. Otherwise, at most n matches are ret
209 // they may not be successive. The result is nil if there ar
210 // or if n == 0.
211 //
212 func (x *Index) FindAllIndex(r *regexp.Regexp, n int) (resul
213     // a non-empty literal prefix is used to determine p
214     // match start indices with Lookup
215     prefix, complete := r.LiteralPrefix()
216     lit := []byte(prefix)
217
218     // worst-case scenario: no literal prefix
219     if prefix == "" {
220         return r.FindAllIndex(x.data, n)
221     }
222
223     // if regexp is a literal just use Lookup and conver
224     // result into match pairs
225     if complete {
226         // Lookup returns indices that may belong to
227         // After eliminating them, we may end up wit
228         // If we don't have enough at the end, redo
229         // increased value n1, but only if Lookup re
230         // indices in the first place (if it returne
231         // there cannot be more).
232         for n1 := n; ; n1 += 2 * (n - len(result)) /
233             indices := x.Lookup(lit, n1)
234             if len(indices) == 0 {
235                 return
236             }
237             sort.Ints(indices)
238             pairs := make([]int, 2*len(indices))
239             result = make([][]int, len(indices))

```

```

240         count := 0
241         prev := 0
242         for _, i := range indices {
243             if count == n {
244                 break
245             }
246             // ignore indices leading to
247             if prev <= i {
248                 j := 2 * count
249                 pairs[j+0] = i
250                 pairs[j+1] = i + len
251                 result[count] = pair
252                 count++
253                 prev = i + len(lit)
254             }
255         }
256         result = result[0:count]
257         if len(result) >= n || len(indices)
258             // found all matches or ther
259             // (n and n1 can be negative
260             break
261         }
262     }
263     if len(result) == 0 {
264         result = nil
265     }
266     return
267 }
268
269 // regexp has a non-empty literal prefix; Lookup(lit
270 // the indices of possible complete matches; use the
271 // points for anchored searches
272 // (regexp "^" matches beginning of input, not begin
273 r = regexp.MustCompile("^" + r.String()) // compiles
274
275 // same comment about Lookup applies here as in the
276 for n1 := n; ; n1 += 2 * (n - len(result)) /* overfl
277     indices := x.Lookup(lit, n1)
278     if len(indices) == 0 {
279         return
280     }
281     sort.Ints(indices)
282     result = result[0:0]
283     prev := 0
284     for _, i := range indices {
285         if len(result) == n {
286             break
287         }
288         m := r.FindIndex(x.data[i:]) // anch

```

```

289             // ignore indices leading to overlap
290             if m != nil && prev <= i {
291                 m[0] = i // correct m
292                 m[1] += i
293                 result = append(result, m)
294                 prev = m[1]
295             }
296         }
297         if len(result) >= n || len(indices) != n1 {
298             // found all matches or there's no c
299             // (n and n1 can be negative)
300             break
301         }
302     }
303     if len(result) == 0 {
304         result = nil
305     }
306     return
307 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/io/io.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package io provides basic interfaces to I/O primitives.
6 // Its primary job is to wrap existing implementations of su
7 // such as those in package os, into shared public interface
8 // abstract the functionality, plus some other related primi
9 //
10 // Because these interfaces and primitives wrap lower-level
11 // various implementations, unless otherwise informed client
12 // assume they are safe for parallel execution.
13 package io
14
15 import (
16     "errors"
17 )
18
19 // ErrShortWrite means that a write accepted fewer bytes tha
20 // but failed to return an explicit error.
21 var ErrShortWrite = errors.New("short write")
22
23 // ErrShortBuffer means that a read required a longer buffer
24 var ErrShortBuffer = errors.New("short buffer")
25
26 // EOF is the error returned by Read when no more input is a
27 // Functions should return EOF only to signal a graceful end
28 // If the EOF occurs unexpectedly in a structured data strea
29 // the appropriate error is either ErrUnexpectedEOF or some
30 // giving more detail.
31 var EOF = errors.New("EOF")
32
33 // ErrUnexpectedEOF means that EOF was encountered in the
34 // middle of reading a fixed-size block or data structure.
35 var ErrUnexpectedEOF = errors.New("unexpected EOF")
36
37 // Reader is the interface that wraps the basic Read method.
38 //
39 // Read reads up to len(p) bytes into p. It returns the num
40 // read (0 <= n <= len(p)) and any error encountered. Even
41 // returns n < len(p), it may use all of p as scratch space
42 // If some data is available but not len(p) bytes, Read conv
43 // returns what is available instead of waiting for more.
44 //
```

```

45 // When Read encounters an error or end-of-file condition af
46 // successfully reading n > 0 bytes, it returns the number o
47 // bytes read. It may return the (non-nil) error from the s
48 // or return the error (and n == 0) from a subsequent call.
49 // An instance of this general case is that a Reader returni
50 // a non-zero number of bytes at the end of the input stream
51 // return either err == EOF or err == nil. The next Read sh
52 // return 0, EOF regardless.
53 //
54 // Callers should always process the n > 0 bytes returned be
55 // considering the error err. Doing so correctly handles I/
56 // that happen after reading some bytes and also both of the
57 // allowed EOF behaviors.
58 type Reader interface {
59     Read(p []byte) (n int, err error)
60 }
61
62 // Writer is the interface that wraps the basic Write method
63 //
64 // Write writes len(p) bytes from p to the underlying data s
65 // It returns the number of bytes written from p (0 <= n <=
66 // and any error encountered that caused the write to stop e
67 // Write must return a non-nil error if it returns n < len(p)
68 type Writer interface {
69     Write(p []byte) (n int, err error)
70 }
71
72 // Closer is the interface that wraps the basic Close method
73 type Closer interface {
74     Close() error
75 }
76
77 // Seeker is the interface that wraps the basic Seek method.
78 //
79 // Seek sets the offset for the next Read or Write to offset
80 // interpreted according to whence: 0 means relative to the
81 // the file, 1 means relative to the current offset, and 2 m
82 // relative to the end. Seek returns the new offset and an
83 // any.
84 type Seeker interface {
85     Seek(offset int64, whence int) (ret int64, err error)
86 }
87
88 // ReadWriter is the interface that groups the basic Read an
89 type ReadWriter interface {
90     Reader
91     Writer
92 }
93
94 // ReadCloser is the interface that groups the basic Read an

```

```

95 type ReadCloser interface {
96     Reader
97     Closer
98 }
99
100 // WriteCloser is the interface that groups the basic Write
101 type WriteCloser interface {
102     Writer
103     Closer
104 }
105
106 // ReadWriteCloser is the interface that groups the basic Re
107 type ReadWriteCloser interface {
108     Reader
109     Writer
110     Closer
111 }
112
113 // ReadSeeker is the interface that groups the basic Read an
114 type ReadSeeker interface {
115     Reader
116     Seeker
117 }
118
119 // WriteSeeker is the interface that groups the basic Write
120 type WriteSeeker interface {
121     Writer
122     Seeker
123 }
124
125 // ReadWriteSeeker is the interface that groups the basic Re
126 type ReadWriteSeeker interface {
127     Reader
128     Writer
129     Seeker
130 }
131
132 // ReaderFrom is the interface that wraps the ReadFrom metho
133 type ReaderFrom interface {
134     ReadFrom(r Reader) (n int64, err error)
135 }
136
137 // WriterTo is the interface that wraps the WriteTo method.
138 type WriterTo interface {
139     WriteTo(w Writer) (n int64, err error)
140 }
141
142 // ReaderAt is the interface that wraps the basic ReadAt met
143 //

```

```

144 // ReadAt reads len(p) bytes into p starting at offset off i
145 // underlying input source. It returns the number of bytes
146 // read (0 <= n <= len(p)) and any error encountered.
147 //
148 // When ReadAt returns n < len(p), it returns a non-nil erro
149 // explaining why more bytes were not returned. In this res
150 // ReadAt is stricter than Read.
151 //
152 // Even if ReadAt returns n < len(p), it may use all of p as
153 // space during the call. If some data is available but not
154 // ReadAt blocks until either all the data is available or a
155 // In this respect ReadAt is different from Read.
156 //
157 // If the n = len(p) bytes returned by ReadAt are at the end
158 // input source, ReadAt may return either err == EOF or err
159 //
160 // If ReadAt is reading from an input source with a seek off
161 // ReadAt should not affect nor be affected by the underlyin
162 // seek offset.
163 //
164 // Clients of ReadAt can execute parallel ReadAt calls on th
165 // same input source.
166 type ReaderAt interface {
167     ReadAt(p []byte, off int64) (n int, err error)
168 }
169
170 // WriterAt is the interface that wraps the basic WriteAt me
171 //
172 // WriteAt writes len(p) bytes from p to the underlying data
173 // at offset off. It returns the number of bytes written fr
174 // and any error encountered that caused the write to stop e
175 // WriteAt must return a non-nil error if it returns n < len
176 //
177 // If WriteAt is writing to a destination with a seek offset
178 // WriteAt should not affect nor be affected by the underlyi
179 // seek offset.
180 //
181 // Clients of WriteAt can execute parallel WriteAt calls on
182 // destination if the ranges do not overlap.
183 type WriterAt interface {
184     WriteAt(p []byte, off int64) (n int, err error)
185 }
186
187 // ByteReader is the interface that wraps the ReadByte metho
188 //
189 // ReadByte reads and returns the next byte from the input.
190 // If no byte is available, err will be set.
191 type ByteReader interface {
192     ReadByte() (c byte, err error)

```

```

193 }
194
195 // ByteScanner is the interface that adds the UnreadByte met
196 // basic ReadByte method.
197 //
198 // UnreadByte causes the next call to ReadByte to return the
199 // as the previous call to ReadByte.
200 // It may be an error to call UnreadByte twice without an in
201 // call to ReadByte.
202 type ByteScanner interface {
203     ByteReader
204     UnreadByte() error
205 }
206
207 // RuneReader is the interface that wraps the ReadRune metho
208 //
209 // ReadRune reads a single UTF-8 encoded Unicode character
210 // and returns the rune and its size in bytes. If no charact
211 // available, err will be set.
212 type RuneReader interface {
213     ReadRune() (r rune, size int, err error)
214 }
215
216 // RuneScanner is the interface that adds the UnreadRune met
217 // basic ReadRune method.
218 //
219 // UnreadRune causes the next call to ReadRune to return the
220 // as the previous call to ReadRune.
221 // It may be an error to call UnreadRune twice without an in
222 // call to ReadRune.
223 type RuneScanner interface {
224     RuneReader
225     UnreadRune() error
226 }
227
228 // stringWriter is the interface that wraps the WriteString
229 type stringWriter interface {
230     WriteString(s string) (n int, err error)
231 }
232
233 // WriteString writes the contents of the string s to w, whi
234 // If w already implements a WriteString method, it is invok
235 func WriteString(w Writer, s string) (n int, err error) {
236     if sw, ok := w.(stringWriter); ok {
237         return sw.WriteString(s)
238     }
239     return w.Write([]byte(s))
240 }
241
242 // ReadAtLeast reads from r into buf until it has read at le

```

```

243 // It returns the number of bytes copied and an error if few
244 // The error is EOF only if no bytes were read.
245 // If an EOF happens after reading fewer than min bytes,
246 // ReadAtLeast returns ErrUnexpectedEOF.
247 // If min is greater than the length of buf, ReadAtLeast ret
248 func ReadAtLeast(r Reader, buf []byte, min int) (n int, err
249     if len(buf) < min {
250         return 0, ErrShortBuffer
251     }
252     for n < min && err == nil {
253         var nn int
254         nn, err = r.Read(buf[n:])
255         n += nn
256     }
257     if err == EOF {
258         if n >= min {
259             err = nil
260         } else if n > 0 {
261             err = ErrUnexpectedEOF
262         }
263     }
264     return
265 }
266
267 // ReadFull reads exactly len(buf) bytes from r into buf.
268 // It returns the number of bytes copied and an error if few
269 // The error is EOF only if no bytes were read.
270 // If an EOF happens after reading some but not all the byte
271 // ReadFull returns ErrUnexpectedEOF.
272 func ReadFull(r Reader, buf []byte) (n int, err error) {
273     return ReadAtLeast(r, buf, len(buf))
274 }
275
276 // CopyN copies n bytes (or until an error) from src to dst.
277 // It returns the number of bytes copied and the earliest
278 // error encountered while copying. Because Read can
279 // return the full amount requested as well as an error
280 // (including EOF), so can CopyN.
281 //
282 // If dst implements the ReaderFrom interface,
283 // the copy is implemented using it.
284 func CopyN(dst Writer, src Reader, n int64) (written int64,
285     // If the writer has a ReadFrom method, use it to do
286     // Avoids a buffer allocation and a copy.
287     if rt, ok := dst.(ReaderFrom); ok {
288         written, err = rt.ReadFrom(LimitReader(src,
289             if written < n && err == nil {
290                 // rt stopped early; must have been
291                 err = EOF

```

```

292         }
293         return
294     }
295     buf := make([]byte, 32*1024)
296     for written < n {
297         l := len(buf)
298         if d := n - written; d < int64(l) {
299             l = int(d)
300         }
301         nr, er := src.Read(buf[0:l])
302         if nr > 0 {
303             nw, ew := dst.Write(buf[0:nr])
304             if nw > 0 {
305                 written += int64(nw)
306             }
307             if ew != nil {
308                 err = ew
309                 break
310             }
311             if nr != nw {
312                 err = ErrShortWrite
313                 break
314             }
315         }
316         if er != nil {
317             err = er
318             break
319         }
320     }
321     return written, err
322 }
323
324 // Copy copies from src to dst until either EOF is reached
325 // on src or an error occurs. It returns the number of byte
326 // copied and the first error encountered while copying, if
327 //
328 // A successful Copy returns err == nil, not err == EOF.
329 // Because Copy is defined to read from src until EOF, it do
330 // not treat an EOF from Read as an error to be reported.
331 //
332 // If dst implements the ReaderFrom interface,
333 // the copy is implemented by calling dst.ReadFrom(src).
334 // Otherwise, if src implements the WriterTo interface,
335 // the copy is implemented by calling src.WriteTo(dst).
336 func Copy(dst Writer, src Reader) (written int64, err error)
337     // If the writer has a ReadFrom method, use it to do
338     // Avoids an allocation and a copy.
339     if rt, ok := dst.(ReaderFrom); ok {
340         return rt.ReadFrom(src)

```

```

341     }
342     // Similarly, if the reader has a WriteTo method, us
343     if wt, ok := src.(WriterTo); ok {
344         return wt.WriteTo(dst)
345     }
346     buf := make([]byte, 32*1024)
347     for {
348         nr, er := src.Read(buf)
349         if nr > 0 {
350             nw, ew := dst.Write(buf[0:nr])
351             if nw > 0 {
352                 written += int64(nw)
353             }
354             if ew != nil {
355                 err = ew
356                 break
357             }
358             if nr != nw {
359                 err = ErrShortWrite
360                 break
361             }
362         }
363         if er == EOF {
364             break
365         }
366         if er != nil {
367             err = er
368             break
369         }
370     }
371     return written, err
372 }
373
374 // LimitReader returns a Reader that reads from r
375 // but stops with EOF after n bytes.
376 // The underlying implementation is a *LimitedReader.
377 func LimitReader(r Reader, n int64) Reader { return &Limited
378
379 // A LimitedReader reads from R but limits the amount of
380 // data returned to just N bytes. Each call to Read
381 // updates N to reflect the new amount remaining.
382 type LimitedReader struct {
383     R Reader // underlying reader
384     N int64  // max bytes remaining
385 }
386
387 func (l *LimitedReader) Read(p []byte) (n int, err error) {
388     if l.N <= 0 {
389         return 0, EOF
390     }

```

```

391         if int64(len(p)) > l.N {
392             p = p[0:l.N]
393         }
394         n, err = l.R.Read(p)
395         l.N -= int64(n)
396         return
397     }
398
399     // NewSectionReader returns a SectionReader that reads from
400     // starting at offset off and stops with EOF after n bytes.
401     func NewSectionReader(r ReaderAt, off int64, n int64) *SectionReader {
402         return &SectionReader{r, off, off, off + n}
403     }
404
405     // SectionReader implements Read, Seek, and ReadAt on a section
406     // of an underlying ReaderAt.
407     type SectionReader struct {
408         r      ReaderAt
409         base   int64
410         off    int64
411         limit  int64
412     }
413
414     func (s *SectionReader) Read(p []byte) (n int, err error) {
415         if s.off >= s.limit {
416             return 0, EOF
417         }
418         if max := s.limit - s.off; int64(len(p)) > max {
419             p = p[0:max]
420         }
421         n, err = s.r.ReadAt(p, s.off)
422         s.off += int64(n)
423         return
424     }
425
426     var errWhence = errors.New("Seek: invalid whence")
427     var errOffset = errors.New("Seek: invalid offset")
428
429     func (s *SectionReader) Seek(offset int64, whence int) (ret int64, err error) {
430         switch whence {
431         default:
432             return 0, errWhence
433         case 0:
434             offset += s.base
435         case 1:
436             offset += s.off
437         case 2:
438             offset += s.limit
439         }

```

```

440         if offset < s.base || offset > s.limit {
441             return 0, errOffset
442         }
443         s.off = offset
444         return offset - s.base, nil
445     }
446
447     func (s *SectionReader) ReadAt(p []byte, off int64) (n int,
448         if off < 0 || off >= s.limit-s.base {
449             return 0, EOF
450         }
451         off += s.base
452         if max := s.limit - off; int64(len(p)) > max {
453             p = p[0:max]
454         }
455         return s.r.ReadAt(p, off)
456     }
457
458     // Size returns the size of the section in bytes.
459     func (s *SectionReader) Size() int64 { return s.limit - s.ba
460
461     // TeeReader returns a Reader that writes to w what it reads
462     // All reads from r performed through it are matched with
463     // corresponding writes to w. There is no internal bufferin
464     // the write must complete before the read completes.
465     // Any error encountered while writing is reported as a read
466     func TeeReader(r Reader, w Writer) Reader {
467         return &teeReader{r, w}
468     }
469
470     type teeReader struct {
471         r Reader
472         w Writer
473     }
474
475     func (t *teeReader) Read(p []byte) (n int, err error) {
476         n, err = t.r.Read(p)
477         if n > 0 {
478             if n, err := t.w.Write(p[:n]); err != nil {
479                 return n, err
480             }
481         }
482         return
483     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/io/multi.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package io
6
7 type multiReader struct {
8     readers []Reader
9 }
10
11 func (mr *multiReader) Read(p []byte) (n int, err error) {
12     for len(mr.readers) > 0 {
13         n, err = mr.readers[0].Read(p)
14         if n > 0 || err != EOF {
15             if err == EOF {
16                 // Don't return EOF yet. The
17                 // in the remaining readers.
18                 err = nil
19             }
20             return
21         }
22         mr.readers = mr.readers[1:]
23     }
24     return 0, EOF
25 }
26
27 // MultiReader returns a Reader that's the logical concatenation
28 // of the provided input readers. They're read sequentially.
29 // Inputs are drained, Read will return EOF.
30 func MultiReader(readers ...Reader) Reader {
31     return &multiReader{readers}
32 }
33
34 type multiWriter struct {
35     writers []Writer
36 }
37
38 func (t *multiWriter) Write(p []byte) (n int, err error) {
39     for _, w := range t.writers {
40         n, err = w.Write(p)
41         if err != nil {
42             return
43         }
44         if n != len(p) {
```

```
45             err = ErrShortWrite
46             return
47         }
48     }
49     return len(p), nil
50 }
51
52 // MultiWriter creates a writer that duplicates its writes t
53 // provided writers, similar to the Unix tee(1) command.
54 func MultiWriter(writers ...Writer) Writer {
55     return &multiWriter{writers}
56 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/io/pipe.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Pipe adapter to connect code expecting an io.Reader
6 // with code expecting an io.Writer.
7
8 package io
9
10 import (
11     "errors"
12     "sync"
13 )
14
15 // ErrClosedPipe is the error used for read or write operati
16 var ErrClosedPipe = errors.New("io: read/write on closed pip
17
18 type pipeResult struct {
19     n    int
20     err error
21 }
22
23 // A pipe is the shared pipe structure underlying PipeReader
24 type pipe struct {
25     r1    sync.Mutex // gates readers one at a time
26     w1    sync.Mutex // gates writers one at a time
27     l     sync.Mutex // protects remaining fields
28     data []byte     // data remaining in pending write
29     rwait sync.Cond  // waiting reader
30     wwait sync.Cond  // waiting writer
31     rerr  error     // if reader closed, error to give
32     werr  error     // if writer closed, error to give
33 }
34
35 func (p *pipe) read(b []byte) (n int, err error) {
36     // One reader at a time.
37     p.r1.Lock()
38     defer p.r1.Unlock()
39
40     p.l.Lock()
41     defer p.l.Unlock()
42     for {
43         if p.rerr != nil {
44             return 0, ErrClosedPipe
```

```

45         }
46         if p.data != nil {
47             break
48         }
49         if p.werr != nil {
50             return 0, p.werr
51         }
52         p.rwait.Wait()
53     }
54     n = copy(b, p.data)
55     p.data = p.data[n:]
56     if len(p.data) == 0 {
57         p.data = nil
58         p.wwait.Signal()
59     }
60     return
61 }
62
63 var zero [0]byte
64
65 func (p *pipe) write(b []byte) (n int, err error) {
66     // pipe uses nil to mean not available
67     if b == nil {
68         b = zero[:]
69     }
70
71     // One writer at a time.
72     p.wl.Lock()
73     defer p.wl.Unlock()
74
75     p.l.Lock()
76     defer p.l.Unlock()
77     p.data = b
78     p.rwait.Signal()
79     for {
80         if p.data == nil {
81             break
82         }
83         if p.rerr != nil {
84             err = p.rerr
85             break
86         }
87         if p.werr != nil {
88             err = ErrClosedPipe
89         }
90         p.wwait.Wait()
91     }
92     n = len(b) - len(p.data)
93     p.data = nil // in case of rerr or werr
94     return

```

```

95 }
96
97 func (p *pipe) rclose(err error) {
98     if err == nil {
99         err = ErrClosedPipe
100    }
101    p.l.Lock()
102    defer p.l.Unlock()
103    p.rerr = err
104    p.rwait.Signal()
105    p.wwait.Signal()
106 }
107
108 func (p *pipe) wclose(err error) {
109     if err == nil {
110         err = EOF
111    }
112    p.l.Lock()
113    defer p.l.Unlock()
114    p.werr = err
115    p.rwait.Signal()
116    p.wwait.Signal()
117 }
118
119 // A PipeReader is the read half of a pipe.
120 type PipeReader struct {
121     p *pipe
122 }
123
124 // Read implements the standard Read interface:
125 // it reads data from the pipe, blocking until a writer
126 // arrives or the write end is closed.
127 // If the write end is closed with an error, that error is
128 // returned as err; otherwise err is EOF.
129 func (r *PipeReader) Read(data []byte) (n int, err error) {
130     return r.p.read(data)
131 }
132
133 // Close closes the reader; subsequent writes to the
134 // write half of the pipe will return the error ErrClosedPip
135 func (r *PipeReader) Close() error {
136     return r.CloseWithError(nil)
137 }
138
139 // CloseWithError closes the reader; subsequent writes
140 // to the write half of the pipe will return the error err.
141 func (r *PipeReader) CloseWithError(err error) error {
142     r.p.rclose(err)
143     return nil

```

```

144 }
145
146 // A PipeWriter is the write half of a pipe.
147 type PipeWriter struct {
148     p *pipe
149 }
150
151 // Write implements the standard Write interface:
152 // it writes data to the pipe, blocking until readers
153 // have consumed all the data or the read end is closed.
154 // If the read end is closed with an error, that err is
155 // returned as err; otherwise err is ErrClosedPipe.
156 func (w *PipeWriter) Write(data []byte) (n int, err error) {
157     return w.p.write(data)
158 }
159
160 // Close closes the writer; subsequent reads from the
161 // read half of the pipe will return no bytes and EOF.
162 func (w *PipeWriter) Close() error {
163     return w.CloseWithError(nil)
164 }
165
166 // CloseWithError closes the writer; subsequent reads from t
167 // read half of the pipe will return no bytes and the error
168 func (w *PipeWriter) CloseWithError(err error) error {
169     w.p.wclose(err)
170     return nil
171 }
172
173 // Pipe creates a synchronous in-memory pipe.
174 // It can be used to connect code expecting an io.Reader
175 // with code expecting an io.Writer.
176 // Reads on one end are matched with writes on the other,
177 // copying data directly between the two; there is no intern
178 // It is safe to call Read and Write in parallel with each o
179 // Close. Close will complete once pending I/O is done. Para
180 // Read, and parallel calls to Write, are also safe:
181 // the individual calls will be gated sequentially.
182 func Pipe() (*PipeReader, *PipeWriter) {
183     p := new(pipe)
184     p.rwait.L = &p.l
185     p.wwait.L = &p.l
186     r := &PipeReader{p}
187     w := &PipeWriter{p}
188     return r, w
189 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/io/ioutil/ioutil.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package ioutil implements some I/O utility functions.
6 package ioutil
7
8 import (
9     "bytes"
10    "io"
11    "os"
12    "sort"
13 )
14
15 // readAll reads from r until an error or EOF and returns th
16 // from the internal buffer allocated with a specified capac
17 func readAll(r io.Reader, capacity int64) (b []byte, err err
18     buf := bytes.NewBuffer(make([]byte, 0, capacity))
19     // If the buffer overflows, we will get bytes.ErrToo
20     // Return that as an error. Any other panic remains.
21     defer func() {
22         e := recover()
23         if e == nil {
24             return
25         }
26         if panicErr, ok := e.(error); ok && panicErr
27             err = panicErr
28         } else {
29             panic(e)
30         }
31     }()
32     _, err = buf.ReadFrom(r)
33     return buf.Bytes(), err
34 }
35
36 // ReadAll reads from r until an error or EOF and returns th
37 // A successful call returns err == nil, not err == EOF. Bec
38 // defined to read from src until EOF, it does not treat an
39 // as an error to be reported.
40 func ReadAll(r io.Reader) ([]byte, error) {
41     return readAll(r, bytes.MinRead)
42 }
43
44 // ReadFile reads the file named by filename and returns the
```

```

45 // A successful call returns err == nil, not err == EOF. Bec
46 // reads the whole file, it does not treat an EOF from Read
47 // to be reported.
48 func ReadFile(filename string) ([]byte, error) {
49     f, err := os.Open(filename)
50     if err != nil {
51         return nil, err
52     }
53     defer f.Close()
54     // It's a good but not certain bet that FileInfo wil
55     // read, so let's try it but be prepared for the ans
56     var n int64
57
58     if fi, err := f.Stat(); err == nil {
59         // Don't preallocate a huge buffer, just in
60         if size := fi.Size(); size < 1e9 {
61             n = size
62         }
63     }
64     // As initial capacity for readAll, use n + a little
65     // and to avoid another allocation after Read has fi
66     // call will read into its allocated internal buffer
67     // wrong, we'll either waste some space off the end
68     // in the overwhelmingly common case we'll get it ju
69     return readAll(f, n+bytes.MinRead)
70 }
71
72 // WriteFile writes data to a file named by filename.
73 // If the file does not exist, WriteFile creates it with per
74 // otherwise WriteFile truncates it before writing.
75 func WriteFile(filename string, data []byte, perm os.FileMod
76     f, err := os.OpenFile(filename, os.O_WRONLY|os.O_CRE
77     if err != nil {
78         return err
79     }
80     n, err := f.Write(data)
81     f.Close()
82     if err == nil && n < len(data) {
83         err = io.ErrShortWrite
84     }
85     return err
86 }
87
88 // byName implements sort.Interface.
89 type byName []os.FileInfo
90
91 func (f byName) Len() int           { return len(f) }
92 func (f byName) Less(i, j int) bool { return f[i].Name() < f
93 func (f byName) Swap(i, j int)      { f[i], f[j] = f[j], f[i]
94

```

```

95 // ReadDir reads the directory named by dirname and returns
96 // a list of sorted directory entries.
97 func ReadDir(dirname string) ([]os.FileInfo, error) {
98     f, err := os.Open(dirname)
99     if err != nil {
100         return nil, err
101     }
102     list, err := f.Readdir(-1)
103     f.Close()
104     if err != nil {
105         return nil, err
106     }
107     sort.Sort(byName(list))
108     return list, nil
109 }
110
111 type nopCloser struct {
112     io.Reader
113 }
114
115 func (nopCloser) Close() error { return nil }
116
117 // NopCloser returns a ReadCloser with a no-op Close method
118 // the provided Reader r.
119 func NopCloser(r io.Reader) io.ReadCloser {
120     return nopCloser{r}
121 }
122
123 type devNull int
124
125 // devNull implements ReaderFrom as an optimization so io.Co
126 // ioutil.Discard can avoid doing unnecessary work.
127 var _ io.ReaderFrom = devNull(0)
128
129 func (devNull) Write(p []byte) (int, error) {
130     return len(p), nil
131 }
132
133 var blackHole = make([]byte, 8192)
134
135 func (devNull) ReadFrom(r io.Reader) (n int64, err error) {
136     readSize := 0
137     for {
138         readSize, err = r.Read(blackHole)
139         n += int64(readSize)
140         if err != nil {
141             if err == io.EOF {
142                 return n, nil
143             }

```

```
144             return
145         }
146     }
147     panic("unreachable")
148 }
149
150 // Discard is an io.Writer on which all Write calls succeed
151 // without doing anything.
152 var Discard io.Writer = devNull(0)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/io/ioutil/tempfile.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package ioutil
6
7 import (
8     "os"
9     "path/filepath"
10    "strconv"
11    "time"
12 )
13
14 // Random number state, accessed without lock; racy but harmless
15 // We generate random temporary file names so that there's a
16 // chance the file doesn't exist yet - keeps the number of
17 // TempFile to a minimum.
18 var rand uint32
19
20 func reseed() uint32 {
21     return uint32(time.Now().UnixNano() + int64(os.Getpid()))
22 }
23
24 func nextSuffix() string {
25     r := rand
26     if r == 0 {
27         r = reseed()
28     }
29     r = r*1664525 + 1013904223 // constants from Numeric
30     rand = r
31     return strconv.Itoa(int(1e9 + r%1e9))[1:]
32 }
33
34 // TempFile creates a new temporary file in the directory dir
35 // with a name beginning with prefix, opens the file for reading
36 // and writing, and returns the resulting *os.File.
37 // If dir is the empty string, TempFile uses the default directory
38 // for temporary files (see os.TempDir).
39 // Multiple programs calling TempFile simultaneously
40 // will not choose the same file. The caller can use f.Name()
41 // to find the name of the file. It is the caller's responsibility
```

```

42 // remove the file when no longer needed.
43 func TempFile(dir, prefix string) (f *os.File, err error) {
44     if dir == "" {
45         dir = os.TempDir()
46     }
47
48     nconflict := 0
49     for i := 0; i < 10000; i++ {
50         name := filepath.Join(dir, prefix+nextSuffix
51         f, err = os.OpenFile(name, os.O_RDWR|os.O_CR
52         if os.IsExist(err) {
53             if nconflict++; nconflict > 10 {
54                 rand = reseed()
55             }
56             continue
57         }
58         break
59     }
60     return
61 }
62
63 // TempDir creates a new temporary directory in the director
64 // with a name beginning with prefix and returns the path of
65 // new directory. If dir is the empty string, TempDir uses
66 // default directory for temporary files (see os.TempDir).
67 // Multiple programs calling TempDir simultaneously
68 // will not choose the same directory. It is the caller's r
69 // to remove the directory when no longer needed.
70 func TempDir(dir, prefix string) (name string, err error) {
71     if dir == "" {
72         dir = os.TempDir()
73     }
74
75     nconflict := 0
76     for i := 0; i < 10000; i++ {
77         try := filepath.Join(dir, prefix+nextSuffix(
78         err = os.Mkdir(try, 0700)
79         if os.IsExist(err) {
80             if nconflict++; nconflict > 10 {
81                 rand = reseed()
82             }
83             continue
84         }
85         if err == nil {
86             name = try
87         }
88         break
89     }
90     return
91 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/log/log.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package log implements a simple logging package. It defin
6 // with methods for formatting output. It also has a predefi
7 // Logger accessible through helper functions Print[f|ln], F
8 // Panic[f|ln], which are easier to use than creating a Logg
9 // That logger writes to standard error and prints the date
10 // of each logged message.
11 // The Fatal functions call os.Exit(1) after writing the log
12 // The Panic functions call panic after writing the log mess
13 package log
14
15 import (
16     "fmt"
17     "io"
18     "os"
19     "runtime"
20     "sync"
21     "time"
22 )
23
24 // These flags define which text to prefix to each log entry
25 const (
26     // Bits or'ed together to control what's printed. Th
27     // order they appear (the order listed here) or the
28     // described in the comments). A colon appears afte
29     //      2009/0123 01:23:23.123123 /a/b/c/d.go:23: me
30     Ldate          = 1 << iota // the date: 2009/01/2
31     Ltime          // the time: 01:23:23
32     Lmicroseconds // microsecond resolut
33     Llongfile     // full file name and
34     Lshortfile    // final file name ele
35     LstdFlags     = Ldate | Ltime // initial values for
36 )
37
38 // A Logger represents an active logging object that generat
39 // output to an io.Writer. Each logging operation makes a s
40 // the Writer's Write method. A Logger can be used simultan
41 // multiple goroutines; it guarantees to serialize access to
42 type Logger struct {
43     mu      sync.Mutex // ensures atomic writes; protects
44     prefix string      // prefix to write at beginning of
```

```

45     flag    int        // properties
46     out     io.Writer  // destination for output
47     buf     []byte     // for accumulating text to write
48 }
49
50 // New creates a new Logger. The out variable sets the
51 // destination to which log data will be written.
52 // The prefix appears at the beginning of each generated log
53 // The flag argument defines the logging properties.
54 func New(out io.Writer, prefix string, flag int) *Logger {
55     return &Logger{out: out, prefix: prefix, flag: flag}
56 }
57
58 var std = New(os.Stderr, "", LstdFlags)
59
60 // Cheap integer to fixed-width decimal ASCII. Give a negat
61 // Knows the buffer has capacity.
62 func itoa(buf *[]byte, i int, wid int) {
63     var u uint = uint(i)
64     if u == 0 && wid <= 1 {
65         *buf = append(*buf, '0')
66         return
67     }
68
69     // Assemble decimal in reverse order.
70     var b [32]byte
71     bp := len(b)
72     for ; u > 0 || wid > 0; u /= 10 {
73         bp--
74         wid--
75         b[bp] = byte(u%10) + '0'
76     }
77     *buf = append(*buf, b[bp:]...)
78 }
79
80 func (l *Logger) formatHeader(buf *[]byte, t time.Time, file
81     *buf = append(*buf, l.prefix...)
82     if l.flag&(Ldate|Ltime|Lmicroseconds) != 0 {
83         if l.flag&Ldate != 0 {
84             year, month, day := t.Date()
85             itoa(buf, year, 4)
86             *buf = append(*buf, '/')
87             itoa(buf, int(month), 2)
88             *buf = append(*buf, '/')
89             itoa(buf, day, 2)
90             *buf = append(*buf, ' ')
91         }
92         if l.flag&(Ltime|Lmicroseconds) != 0 {
93             hour, min, sec := t.Clock()
94             itoa(buf, hour, 2)

```

```

95         *buf = append(*buf, ':')
96         itoa(buf, min, 2)
97         *buf = append(*buf, ':')
98         itoa(buf, sec, 2)
99         if l.flag&Lmicroseconds != 0 {
100             *buf = append(*buf, '.')
101             itoa(buf, t.Nanosecond()/1e3
102         }
103         *buf = append(*buf, ' ')
104     }
105 }
106 if l.flag&(Lshortfile|Llongfile) != 0 {
107     if l.flag&Lshortfile != 0 {
108         short := file
109         for i := len(file) - 1; i > 0; i-- {
110             if file[i] == '/' {
111                 short = file[i+1:]
112                 break
113             }
114         }
115         file = short
116     }
117     *buf = append(*buf, file...)
118     *buf = append(*buf, ':')
119     itoa(buf, line, -1)
120     *buf = append(*buf, ": "...)
121 }
122 }
123
124 // Output writes the output for a logging event. The string
125 // the text to print after the prefix specified by the flags
126 // Logger. A newline is appended if the last character of s
127 // already a newline. Calldepth is used to recover the PC a
128 // provided for generality, although at the moment on all pr
129 // paths it will be 2.
130 func (l *Logger) Output(calldepth int, s string) error {
131     now := time.Now() // get this early.
132     var file string
133     var line int
134     l.mu.Lock()
135     defer l.mu.Unlock()
136     if l.flag&(Lshortfile|Llongfile) != 0 {
137         // release lock while getting caller info -
138         l.mu.Unlock()
139         var ok bool
140         _, file, line, ok = runtime.Caller(calldepth
141         if !ok {
142             file = "???"
143             line = 0

```

```

144         }
145         l.mu.Lock()
146     }
147     l.buf = l.buf[:0]
148     l.formatHeader(&l.buf, now, file, line)
149     l.buf = append(l.buf, s...)
150     if len(s) > 0 && s[len(s)-1] != '\n' {
151         l.buf = append(l.buf, '\n')
152     }
153     _, err := l.out.Write(l.buf)
154     return err
155 }
156
157 // Printf calls l.Output to print to the logger.
158 // Arguments are handled in the manner of fmt.Printf.
159 func (l *Logger) Printf(format string, v ...interface{}) {
160     l.Output(2, fmt.Sprintf(format, v...))
161 }
162
163 // Print calls l.Output to print to the logger.
164 // Arguments are handled in the manner of fmt.Print.
165 func (l *Logger) Print(v ...interface{}) { l.Output(2, fmt.S
166
167 // Println calls l.Output to print to the logger.
168 // Arguments are handled in the manner of fmt.Println.
169 func (l *Logger) Println(v ...interface{}) { l.Output(2, fmt
170
171 // Fatal is equivalent to l.Print() followed by a call to os
172 func (l *Logger) Fatal(v ...interface{}) {
173     l.Output(2, fmt.Sprint(v...))
174     os.Exit(1)
175 }
176
177 //.Fatalf is equivalent to l.Printf() followed by a call to
178 func (l *Logger) Fatalf(format string, v ...interface{}) {
179     l.Output(2, fmt.Sprintf(format, v...))
180     os.Exit(1)
181 }
182
183 // Fatalln is equivalent to l.Println() followed by a call t
184 func (l *Logger) Fatalln(v ...interface{}) {
185     l.Output(2, fmt.Sprintln(v...))
186     os.Exit(1)
187 }
188
189 // Panic is equivalent to l.Print() followed by a call to pa
190 func (l *Logger) Panic(v ...interface{}) {
191     s := fmt.Sprint(v...)
192     l.Output(2, s)

```

```

193         panic(s)
194     }
195
196 // Panicf is equivalent to l.Printf() followed by a call to
197 func (l *Logger) Panicf(format string, v ...interface{}) {
198     s := fmt.Sprintf(format, v...)
199     l.Output(2, s)
200     panic(s)
201 }
202
203 // Panicln is equivalent to l.Println() followed by a call t
204 func (l *Logger) Panicln(v ...interface{}) {
205     s := fmt.Sprintln(v...)
206     l.Output(2, s)
207     panic(s)
208 }
209
210 // Flags returns the output flags for the logger.
211 func (l *Logger) Flags() int {
212     l.mu.Lock()
213     defer l.mu.Unlock()
214     return l.flag
215 }
216
217 // SetFlags sets the output flags for the logger.
218 func (l *Logger) SetFlags(flag int) {
219     l.mu.Lock()
220     defer l.mu.Unlock()
221     l.flag = flag
222 }
223
224 // Prefix returns the output prefix for the logger.
225 func (l *Logger) Prefix() string {
226     l.mu.Lock()
227     defer l.mu.Unlock()
228     return l.prefix
229 }
230
231 // SetPrefix sets the output prefix for the logger.
232 func (l *Logger) SetPrefix(prefix string) {
233     l.mu.Lock()
234     defer l.mu.Unlock()
235     l.prefix = prefix
236 }
237
238 // SetOutput sets the output destination for the standard lo
239 func SetOutput(w io.Writer) {
240     std.mu.Lock()
241     defer std.mu.Unlock()
242     std.out = w

```

```
243 }
244
245 // Flags returns the output flags for the standard logger.
246 func Flags() int {
247     return std.Flags()
248 }
249
250 // SetFlags sets the output flags for the standard logger.
251 func SetFlags(flag int) {
252     std.SetFlags(flag)
253 }
254
255 // Prefix returns the output prefix for the standard logger.
256 func Prefix() string {
257     return std.Prefix()
258 }
259
260 // SetPrefix sets the output prefix for the standard logger.
261 func SetPrefix(prefix string) {
262     std.SetPrefix(prefix)
263 }
264
265 // These functions write to the standard logger.
266
267 // Print calls Output to print to the standard logger.
268 // Arguments are handled in the manner of fmt.Print.
269 func Print(v ...interface{}) {
270     std.Output(2, fmt.Sprint(v...))
271 }
272
273 // Printf calls Output to print to the standard logger.
274 // Arguments are handled in the manner of fmt.Printf.
275 func Printf(format string, v ...interface{}) {
276     std.Output(2, fmt.Sprintf(format, v...))
277 }
278
279 // Println calls Output to print to the standard logger.
280 // Arguments are handled in the manner of fmt.Println.
281 func Println(v ...interface{}) {
282     std.Output(2, fmt.Sprintln(v...))
283 }
284
285 // Fatal is equivalent to Print() followed by a call to os.E
286 func Fatal(v ...interface{}) {
287     std.Output(2, fmt.Sprint(v...))
288     os.Exit(1)
289 }
290
291 // Fatalf is equivalent to Printf() followed by a call to os
```

```

292 func Fata1f(format string, v ...interface{}) {
293     std.Output(2, fmt.Sprintf(format, v...))
294     os.Exit(1)
295 }
296
297 // Fata1ln is equivalent to Println() followed by a call to
298 func Fata1ln(v ...interface{}) {
299     std.Output(2, fmt.Sprintln(v...))
300     os.Exit(1)
301 }
302
303 // Panic is equivalent to Print() followed by a call to pani
304 func Panic(v ...interface{}) {
305     s := fmt.Sprint(v...)
306     std.Output(2, s)
307     panic(s)
308 }
309
310 // Panicf is equivalent to Printf() followed by a call to pa
311 func Panicf(format string, v ...interface{}) {
312     s := fmt.Sprintf(format, v...)
313     std.Output(2, s)
314     panic(s)
315 }
316
317 // Panicln is equivalent to Println() followed by a call to
318 func Panicln(v ...interface{}) {
319     s := fmt.Sprintln(v...)
320     std.Output(2, s)
321     panic(s)
322 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/log/syslog/syslog.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build !windows,!plan9
6
7 // Package syslog provides a simple interface to the system
8 // can send messages to the syslog daemon using UNIX domain
9 // TCP connections.
10 package syslog
11
12 import (
13     "errors"
14     "fmt"
15     "log"
16     "net"
17     "os"
18 )
19
20 type Priority int
21
22 const (
23     // From /usr/include/sys/syslog.h.
24     // These are the same on Linux, BSD, and OS X.
25     LOG_EMERG Priority = iota
26     LOG_ALERT
27     LOG_CRIT
28     LOG_ERR
29     LOG_WARNING
30     LOG_NOTICE
31     LOG_INFO
32     LOG_DEBUG
33 )
34
35 // A Writer is a connection to a syslog server.
36 type Writer struct {
37     priority Priority
38     prefix   string
39     conn     serverConn
40 }
41
```

```

42 type serverConn interface {
43     writeBytes(p Priority, prefix string, b []byte) (int
44     writeString(p Priority, prefix string, s string) (in
45     close() error
46 }
47
48 type netConn struct {
49     conn net.Conn
50 }
51
52 // New establishes a new connection to the system log daemon
53 // Each write to the returned writer sends a log message wit
54 // the given priority and prefix.
55 func New(priority Priority, prefix string) (w *Writer, err e
56     return Dial("", "", priority, prefix)
57 }
58
59 // Dial establishes a connection to a log daemon by connecti
60 // to address raddr on the network net.
61 // Each write to the returned writer sends a log message wit
62 // the given priority and prefix.
63 func Dial(network, raddr string, priority Priority, prefix s
64     if prefix == "" {
65         prefix = os.Args[0]
66     }
67     var conn serverConn
68     if network == "" {
69         conn, err = unixSyslog()
70     } else {
71         var c net.Conn
72         c, err = net.Dial(network, raddr)
73         conn = netConn{c}
74     }
75     return &Writer{priority, prefix, conn}, err
76 }
77
78 // Write sends a log message to the syslog daemon.
79 func (w *Writer) Write(b []byte) (int, error) {
80     if w.priority > LOG_DEBUG || w.priority < LOG_EMERG
81         return 0, errors.New("log/syslog: invalid pr
82     }
83     return w.conn.writeBytes(w.priority, w.prefix, b)
84 }
85
86 func (w *Writer) writeString(p Priority, s string) (int, err
87     return w.conn.writeString(p, w.prefix, s)
88 }
89
90 func (w *Writer) Close() error { return w.conn.close() }
91

```

```
92 // Emerg logs a message using the LOG_EMERG priority.
93 func (w *Writer) Emerg(m string) (err error) {
94     _, err = w.writeString(LOG_EMERG, m)
95     return err
96 }
97
98 // Alert logs a message using the LOG_ALERT priority.
99 func (w *Writer) Alert(m string) (err error) {
100     _, err = w.writeString(LOG_ALERT, m)
101     return err
102 }
103
104 // Crit logs a message using the LOG_CRIT priority.
105 func (w *Writer) Crit(m string) (err error) {
106     _, err = w.writeString(LOG_CRIT, m)
107     return err
108 }
109
110 // Err logs a message using the LOG_ERR priority.
111 func (w *Writer) Err(m string) (err error) {
112     _, err = w.writeString(LOG_ERR, m)
113     return err
114 }
115
116 // Warning logs a message using the LOG_WARNING priority.
117 func (w *Writer) Warning(m string) (err error) {
118     _, err = w.writeString(LOG_WARNING, m)
119     return err
120 }
121
122 // Notice logs a message using the LOG_NOTICE priority.
123 func (w *Writer) Notice(m string) (err error) {
124     _, err = w.writeString(LOG_NOTICE, m)
125     return err
126 }
127
128 // Info logs a message using the LOG_INFO priority.
129 func (w *Writer) Info(m string) (err error) {
130     _, err = w.writeString(LOG_INFO, m)
131     return err
132 }
133
134 // Debug logs a message using the LOG_DEBUG priority.
135 func (w *Writer) Debug(m string) (err error) {
136     _, err = w.writeString(LOG_DEBUG, m)
137     return err
138 }
139
140 func (n netConn) writeBytes(p Priority, prefix string, b []b
```

```

141         _, err := fmt.Fprintf(n.conn, "<%d>%s: %s\n", p, pre
142         if err != nil {
143             return 0, err
144         }
145         return len(b), nil
146     }
147
148     func (n netConn) writeString(p Priority, prefix string, s st
149         _, err := fmt.Fprintf(n.conn, "<%d>%s: %s\n", p, pre
150         if err != nil {
151             return 0, err
152         }
153         return len(s), nil
154     }
155
156     func (n netConn) close() error {
157         return n.conn.Close()
158     }
159
160     // NewLogger creates a log.Logger whose output is written to
161     // the system log service with the specified priority. The l
162     // argument is the flag set passed through to log.New to cre
163     // the Logger.
164     func NewLogger(p Priority, logFlag int) (*log.Logger, error)
165         s, err := New(p, "")
166         if err != nil {
167             return nil, err
168         }
169         return log.New(s, "", logFlag), nil
170     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/log/syslog/syslog_unix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build !windows,!plan9
6
7 package syslog
8
9 import (
10     "errors"
11     "net"
12 )
13
14 // unixSyslog opens a connection to the syslog daemon running
15 // local machine using a Unix domain socket.
16
17 func unixSyslog() (conn serverConn, err error) {
18     logTypes := []string{"unixgram", "unix"}
19     logPaths := []string{"/dev/log", "/var/run/syslog"}
20     var raddr string
21     for _, network := range logTypes {
22         for _, path := range logPaths {
23             raddr = path
24             conn, err := net.Dial(network, raddr)
25             if err != nil {
26                 continue
27             } else {
28                 return netConn{conn}, nil
29             }
30         }
31     }
32     return nil, errors.New("Unix syslog delivery error")
33 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/abs.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Abs returns the absolute value of x.
8 //
9 // Special cases are:
10 //     Abs(±Inf) = ±Inf
11 //     Abs(NaN) = NaN
12 func Abs(x float64) float64
13
14 func abs(x float64) float64 {
15     switch {
16     case x < 0:
17         return -x
18     case x == 0:
19         return 0 // return correctly abs(-0)
20     }
21     return x
22 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/acosh.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // The original C code, the long comment, and the constants
8 // below are from FreeBSD's /usr/src/lib/msun/src/e_acosh.c
9 // and came with this notice. The go code is a simplified
10 // version of the original C.
11 //
12 // =====
13 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights r
14 //
15 // Developed at SunPro, a Sun Microsystems, Inc. business.
16 // Permission to use, copy, modify, and distribute this
17 // software is freely granted, provided that this notice
18 // is preserved.
19 // =====
20 //
21 //
22 // __ieee754_acosh(x)
23 // Method :
24 //     Based on
25 //         acosh(x) = log [ x + sqrt(x*x-1) ]
26 //     we have
27 //         acosh(x) := log(x)+ln2, if x is large; else
28 //         acosh(x) := log(2x-1/(sqrt(x*x-1)+x)) if x>2
29 //         acosh(x) := log1p(t+sqrt(2.0*t+t*t)); where
30 //
31 // Special cases:
32 //     acosh(x) is NaN with signal if x<1.
33 //     acosh(NaN) is NaN without signal.
34 //
35
36 // Acosh(x) calculates the inverse hyperbolic cosine of x.
37 //
38 // Special cases are:
39 //     Acosh(+Inf) = +Inf
40 //     Acosh(x) = NaN if x < 1
41 //     Acosh(NaN) = NaN
42 func Acosh(x float64) float64 {
43     const (
44         Ln2      = 6.93147180559945286227e-01 // 0x3FE6
```

```

45         Large = 1 << 28                               // 2**28
46     )
47     // first case is special case
48     switch {
49     case x < 1 || IsNaN(x):
50         return NaN()
51     case x == 1:
52         return 0
53     case x >= Large:
54         return Log(x) + Ln2 // x > 2**28
55     case x > 2:
56         return Log(2*x - 1/(x+Sqrt(x*x-1))) // 2**28
57     }
58     t := x - 1
59     return Log1p(t + Sqrt(2*t+t*t)) // 2 >= x > 1
60 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/asin.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Floating-point arcsine and arccosine.
9
10    They are implemented by computing the arctangent
11    after appropriate range reduction.
12 */
13
14 // Asin returns the arcsine of x.
15 //
16 // Special cases are:
17 //     Asin( $\pm 0$ ) =  $\pm 0$ 
18 //     Asin(x) = NaN if  $x < -1$  or  $x > 1$ 
19 func Asin(x float64) float64
20
21 func asin(x float64) float64 {
22     if x == 0 {
23         return x // special case
24     }
25     sign := false
26     if x < 0 {
27         x = -x
28         sign = true
29     }
30     if x > 1 {
31         return NaN() // special case
32     }
33
34     temp := Sqrt(1 - x*x)
35     if x > 0.7 {
36         temp = Pi/2 - satan(temp/x)
37     } else {
38         temp = satan(x / temp)
39     }
40
41     if sign {
42         temp = -temp
43     }
44     return temp
}
```

```
45 }
46
47 // Acos returns the arccosine of x.
48 //
49 // Special case is:
50 //     Acos(x) = NaN if x < -1 or x > 1
51 func Acos(x float64) float64
52
53 func acos(x float64) float64 {
54     return Pi/2 - Asin(x)
55 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/asinh.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // The original C code, the long comment, and the constants
8 // below are from FreeBSD's /usr/src/lib/msun/src/s_asinh.c
9 // and came with this notice. The go code is a simplified
10 // version of the original C.
11 //
12 // =====
13 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights r
14 //
15 // Developed at SunPro, a Sun Microsystems, Inc. business.
16 // Permission to use, copy, modify, and distribute this
17 // software is freely granted, provided that this notice
18 // is preserved.
19 // =====
20 //
21 //
22 // asinh(x)
23 // Method :
24 //     Based on
25 //         asinh(x) = sign(x) * log [ |x| + sqrt(x*x+1)
26 //     we have
27 //         asinh(x) := x if 1+x*x=1,
28 //                 := sign(x)*(log(x)+ln2)) for large |x|, els
29 //                 := sign(x)*log(2|x|+1/(|x|+sqrt(x*x+1))) if
30 //                 := sign(x)*log1p(|x| + x**2/(1 + sqrt(1+x**
31 //
32
33 // Asinh(x) calculates the inverse hyperbolic sine of x.
34 //
35 // Special cases are:
36 //     Asinh(±0) = ±0
37 //     Asinh(±Inf) = ±Inf
38 //     Asinh(NaN) = NaN
39 func Asinh(x float64) float64 {
40     const (
41         Ln2      = 6.93147180559945286227e-01 // 0x3
42         NearZero = 1.0 / (1 << 28)           // 2**
43         Large    = 1 << 28                     // 2**
44     )
```

```

45     // special cases
46     if IsNaN(x) || IsInf(x, 0) {
47         return x
48     }
49     sign := false
50     if x < 0 {
51         x = -x
52         sign = true
53     }
54     var temp float64
55     switch {
56     case x > Large:
57         temp = Log(x) + Ln2 // |x| > 2**28
58     case x > 2:
59         temp = Log(2*x + 1/(Sqrt(x*x+1)+x)) // 2**28
60     case x < NearZero:
61         temp = x // |x| < 2**-28
62     default:
63         temp = Log1p(x + x*x/(1+Sqrt(1+x*x))) // 2.0
64     }
65     if sign {
66         temp = -temp
67     }
68     return temp
69 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/atan.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Floating-point arctangent.
9
10    Atan returns the value of the arctangent of its
11    argument in the range  $[-\pi/2, \pi/2]$ .
12    There are no error returns.
13    Coefficients are #5077 from Hart & Cheney. (19.56D)
14 */
15
16 // xatan evaluates a series valid in the
17 // range  $[-0.414\dots, +0.414\dots]$ . ( $\tan(\pi/8)$ )
18 func xatan(arg float64) float64 {
19     const (
20         P4 = .161536412982230228262e2
21         P3 = .26842548195503973794141e3
22         P2 = .11530293515404850115428136e4
23         P1 = .178040631643319697105464587e4
24         P0 = .89678597403663861959987488e3
25         Q4 = .5895697050844462222791e2
26         Q3 = .536265374031215315104235e3
27         Q2 = .16667838148816337184521798e4
28         Q1 = .207933497444540981287275926e4
29         Q0 = .89678597403663861962481162e3
30     )
31     sq := arg * arg
32     value := (((P4*sq+P3)*sq+P2)*sq+P1)*sq + P0
33     value = value / (((((sq+Q4)*sq+Q3)*sq+Q2)*sq+Q1)*sq
34     return value * arg
35 }
36
37 // satan reduces its argument (known to be positive)
38 // to the range  $[0, 0.414\dots]$  and calls xatan.
39 func satan(arg float64) float64 {
40     if arg < Sqrt2-1 {
41         return xatan(arg)
42     }
43     if arg > Sqrt2+1 {
44         return Pi/2 - xatan(1/arg)
```

```

45         }
46         return Pi/4 + xatan((arg-1)/(arg+1))
47     }
48
49     // Atan returns the arctangent of x.
50     //
51     // Special cases are:
52     //     Atan(±0) = ±0
53     //     Atan(±Inf) = ±Pi/2
54     func Atan(x float64) float64
55
56     func atan(x float64) float64 {
57         if x == 0 {
58             return x
59         }
60         if x > 0 {
61             return satan(x)
62         }
63         return -satan(-x)
64     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/atan2.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Atan2 returns the arc tangent of y/x, using
8 // the signs of the two to determine the quadrant
9 // of the return value.
10 //
11 // Special cases are (in order):
12 //     Atan2(y, NaN) = NaN
13 //     Atan2(NaN, x) = NaN
14 //     Atan2(+0, x>=0) = +0
15 //     Atan2(-0, x>=0) = -0
16 //     Atan2(+0, x<=-0) = +Pi
17 //     Atan2(-0, x<=-0) = -Pi
18 //     Atan2(y>0, 0) = +Pi/2
19 //     Atan2(y<0, 0) = -Pi/2
20 //     Atan2(+Inf, +Inf) = +Pi/4
21 //     Atan2(-Inf, +Inf) = -Pi/4
22 //     Atan2(+Inf, -Inf) = 3Pi/4
23 //     Atan2(-Inf, -Inf) = -3Pi/4
24 //     Atan2(y, +Inf) = 0
25 //     Atan2(y>0, -Inf) = +Pi
26 //     Atan2(y<0, -Inf) = -Pi
27 //     Atan2(+Inf, x) = +Pi/2
28 //     Atan2(-Inf, x) = -Pi/2
29 func Atan2(y, x float64) float64
30
31 func atan2(y, x float64) float64 {
32     // special cases
33     switch {
34     case IsNaN(y) || IsNaN(x):
35         return NaN()
36     case y == 0:
37         if x >= 0 && !Signbit(x) {
38             return Copysign(0, y)
39         }
40         return Copysign(Pi, y)
41     case x == 0:
42         return Copysign(Pi/2, y)
43     case IsInf(x, 0):
44         if IsInf(x, 1) {
```

```

45         switch {
46         case IsInf(y, 0):
47             return Copysign(Pi/4, y)
48         default:
49             return Copysign(0, y)
50         }
51     }
52     switch {
53     case IsInf(y, 0):
54         return Copysign(3*Pi/4, y)
55     default:
56         return Copysign(Pi, y)
57     }
58     case IsInf(y, 0):
59         return Copysign(Pi/2, y)
60 }
61
62 // Call atan and determine the quadrant.
63 q := Atan(y / x)
64 if x < 0 {
65     if q <= 0 {
66         return q + Pi
67     }
68     return q - Pi
69 }
70 return q
71 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/atanh.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // The original C code, the long comment, and the constants
8 // below are from FreeBSD's /usr/src/lib/msun/src/e_atanh.c
9 // and came with this notice. The go code is a simplified
10 // version of the original C.
11 //
12 // =====
13 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
14 //
15 // Developed at SunPro, a Sun Microsystems, Inc. business.
16 // Permission to use, copy, modify, and distribute this
17 // software is freely granted, provided that this notice
18 // is preserved.
19 // =====
20 //
21 //
22 // __ieee754_atanh(x)
23 // Method :
24 //     1. Reduce x to positive by atanh(-x) = -atanh(x)
25 //     2. For x>=0.5
26 //           1           2x
27 //     atanh(x) = --- * log(1 + -----) = 0.5 * log1p(2 *
28 //                2           1 - x
29 //
30 //     For x<0.5
31 //     atanh(x) = 0.5*log1p(2x+2x*x/(1-x))
32 //
33 // Special cases:
34 //     atanh(x) is NaN if |x| > 1 with signal;
35 //     atanh(NaN) is that NaN with no signal;
36 //     atanh(+/-1) is +/-INF with signal.
37 //
38
39 // Atanh(x) calculates the inverse hyperbolic tangent of x.
40 //
41 // Special cases are:
42 //     Atanh(1) = +Inf
43 //     Atanh(±0) = ±0
44 //     Atanh(-1) = -Inf
```

```

45 //      Atanh(x) = NaN if x < -1 or x > 1
46 //      Atanh(NaN) = NaN
47 func Atanh(x float64) float64 {
48     const NearZero = 1.0 / (1 << 28) // 2**-28
49     // special cases
50     switch {
51     case x < -1 || x > 1 || IsNaN(x):
52         return NaN()
53     case x == 1:
54         return Inf(1)
55     case x == -1:
56         return Inf(-1)
57     }
58     sign := false
59     if x < 0 {
60         x = -x
61         sign = true
62     }
63     var temp float64
64     switch {
65     case x < NearZero:
66         temp = x
67     case x < 0.5:
68         temp = x + x
69         temp = 0.5 * Log1p(temp+temp*x/(1-x))
70     default:
71         temp = 0.5 * Log1p((x+x)/(1-x))
72     }
73     if sign {
74         temp = -temp
75     }
76     return temp
77 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/bits.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 const (
8     uvnan      = 0x7FF0000000000001
9     uvinf      = 0x7FF0000000000000
10    uvneginf   = 0xFFF0000000000000
11    mask       = 0x7FF
12    shift      = 64 - 11 - 1
13    bias       = 1023
14 )
15
16 // Inf returns positive infinity if sign >= 0, negative infi
17 func Inf(sign int) float64 {
18     var v uint64
19     if sign >= 0 {
20         v = uvinf
21     } else {
22         v = uvneginf
23     }
24     return Float64frombits(v)
25 }
26
27 // NaN returns an IEEE 754 ``not-a-number'' value.
28 func NaN() float64 { return Float64frombits(uvnan) }
29
30 // IsNaN returns whether f is an IEEE 754 ``not-a-number'' v
31 func IsNaN(f float64) (is bool) {
32     // IEEE 754 says that only NaNs satisfy f != f.
33     // To avoid the floating-point hardware, could use:
34     //     x := Float64bits(f);
35     //     return uint32(x>>shift)&mask == mask && x !=
36     return f != f
37 }
38
39 // IsInf returns whether f is an infinity, according to sign
40 // If sign > 0, IsInf returns whether f is positive infinity
41 // If sign < 0, IsInf returns whether f is negative infinity
42 // If sign == 0, IsInf returns whether f is either infinity.
43 func IsInf(f float64, sign int) bool {
44     // Test for infinity by comparing against maximum fl
```

```
45         // To avoid the floating-point hardware, could use:
46         //         x := Float64bits(f);
47         //         return sign >= 0 && x == uvinf || sign <= 0
48         return sign >= 0 && f > MaxFloat64 || sign <= 0 && f
49     }
50
51     // normalize returns a normal number y and exponent exp
52     // satisfying x == y * 2**exp. It assumes x is finite and no
53     func normalize(x float64) (y float64, exp int) {
54         const SmallestNormal = 2.2250738585072014e-308 // 2*
55         if Abs(x) < SmallestNormal {
56             return x * (1 << 52), -52
57         }
58         return x, 0
59     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/cbrt.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     The algorithm is based in part on "Optimal Partition
9     Newton's Method for Calculating Roots", by Gunter Me
10    and G. D. Taylor, Mathematics of Computation © 1980
11    Mathematical Society.
12    (http://www.jstor.org/stable/2006387?seq=9, accessed
13 */
14
15 // Cbrt returns the cube root of its argument.
16 //
17 // Special cases are:
18 //     Cbrt(±0) = ±0
19 //     Cbrt(±Inf) = ±Inf
20 //     Cbrt(NaN) = NaN
21 func Cbrt(x float64) float64 {
22     const (
23         A1 = 1.662848358e-01
24         A2 = 1.096040958e+00
25         A3 = 4.105032829e-01
26         A4 = 5.649335816e-01
27         B1 = 2.639607233e-01
28         B2 = 8.699282849e-01
29         B3 = 1.629083358e-01
30         B4 = 2.824667908e-01
31         C1 = 4.190115298e-01
32         C2 = 6.904625373e-01
33         C3 = 6.46502159e-02
34         C4 = 1.412333954e-01
35     )
36     // special cases
37     switch {
38     case x == 0 || IsNaN(x) || IsInf(x, 0):
39         return x
40     }
41     sign := false
42     if x < 0 {
43         x = -x
44         sign = true
```

```

45     }
46     // Reduce argument and estimate cube root
47     f, e := Frexp(x) // 0.5 <= f < 1.0
48     m := e % 3
49     if m > 0 {
50         m -= 3
51         e -= m // e is multiple of 3
52     }
53     switch m {
54     case 0: // 0.5 <= f < 1.0
55         f = A1*f + A2 - A3/(A4+f)
56     case -1:
57         f *= 0.5 // 0.25 <= f < 0.5
58         f = B1*f + B2 - B3/(B4+f)
59     default: // m == -2
60         f *= 0.25 // 0.125 <= f < 0.25
61         f = C1*f + C2 - C3/(C4+f)
62     }
63     y := Ldexp(f, e/3) // e/3 = exponent of cube root
64
65     // Iterate
66     s := y * y * y
67     t := s + x
68     y *= (t + x) / (s + t)
69     // Reiterate
70     s = (y*y*y - x) / x
71     y -= y * (((14.0/81.0)*s-(2.0/9.0))*s + (1.0 / 3.0))
72     if sign {
73         y = -y
74     }
75     return y
76 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/const.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package math provides basic constants and mathematical fu
6 package math
7
8 // Mathematical constants.
9 // Reference: http://oeis.org/Axxxxxx
10 const (
11     E      = 2.71828182845904523536028747135266249775724709
12     Pi     = 3.14159265358979323846264338327950288419716939
13     Phi    = 1.61803398874989484820458683436563811772030917
14
15     Sqrt2   = 1.4142135623730950488016887242096980785696
16     SqrtE   = 1.6487212707001281468486507878141635716537
17     SqrtPi  = 1.7724538509055160272981674833411451827975
18     SqrtPhi = 1.2720196495140689642524224617374914917156
19
20     Ln2     = 0.69314718055994530941723212145817656807550
21     Log2E   = 1 / Ln2
22     Ln10    = 2.30258509299404568401799145468436420760110
23     Log10E  = 1 / Ln10
24 )
25
26 // Floating-point limit values.
27 // Max is the largest finite value representable by the type
28 // SmallestNonzero is the smallest positive, non-zero value
29 const (
30     MaxFloat32           = 3.4028234663852885981170418
31     SmallestNonzeroFloat32 = 1.4012984643248170709237295
32
33     MaxFloat64           = 1.7976931348623157081452742
34     SmallestNonzeroFloat64 = 4.9406564584124654417656879
35 )
36
37 // Integer limit values.
38 const (
39     MaxInt8   = 1<<7 - 1
40     MinInt8   = -1 << 7
41     MaxInt16  = 1<<15 - 1
42     MinInt16  = -1 << 15
43     MaxInt32  = 1<<31 - 1
44     MinInt32  = -1 << 31
```

```
45         MaxInt64  = 1<<63 - 1
46         MinInt64  = -1 << 63
47         MaxUint8   = 1<<8 - 1
48         MaxUint16  = 1<<16 - 1
49         MaxUint32  = 1<<32 - 1
50         MaxUint64  = 1<<64 - 1
51     )
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/copysign.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Copysign(x, y) returns a value with the magnitude
8 // of x and the sign of y.
9 func Copysign(x, y float64) float64 {
10     const sign = 1 << 63
11     return Float64frombits(Float64bits(x)&^sign | Float6
12 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/dim.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Dim returns the maximum of x-y or 0.
8 //
9 // Special cases are:
10 //     Dim(+Inf, +Inf) = NaN
11 //     Dim(-Inf, -Inf) = NaN
12 //     Dim(x, NaN) = Dim(NaN, x) = NaN
13 func Dim(x, y float64) float64
14
15 func dim(x, y float64) float64 {
16     return max(x-y, 0)
17 }
18
19 // Max returns the larger of x or y.
20 //
21 // Special cases are:
22 //     Max(x, +Inf) = Max(+Inf, x) = +Inf
23 //     Max(x, NaN) = Max(NaN, x) = NaN
24 //     Max(+0, ±0) = Max(±0, +0) = +0
25 //     Max(-0, -0) = -0
26 func Max(x, y float64) float64
27
28 func max(x, y float64) float64 {
29     // special cases
30     switch {
31     case IsInf(x, 1) || IsInf(y, 1):
32         return Inf(1)
33     case IsNaN(x) || IsNaN(y):
34         return NaN()
35     case x == 0 && x == y:
36         if Signbit(x) {
37             return y
38         }
39         return x
40     }
41     if x > y {
42         return x
43     }
44     return y

```

```

45 }
46
47 // Min returns the smaller of x or y.
48 //
49 // Special cases are:
50 //     Min(x, -Inf) = Min(-Inf, x) = -Inf
51 //     Min(x, NaN) = Min(NaN, x) = NaN
52 //     Min(-0, ±0) = Min(±0, -0) = -0
53 func Min(x, y float64) float64
54
55 func min(x, y float64) float64 {
56     // special cases
57     switch {
58     case IsInf(x, -1) || IsInf(y, -1):
59         return Inf(-1)
60     case IsNaN(x) || IsNaN(y):
61         return NaN()
62     case x == 0 && x == y:
63         if Signbit(x) {
64             return x
65         }
66         return y
67     }
68     if x < y {
69         return x
70     }
71     return y
72 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/erf.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8      Floating-point error function and complementary error
9 */
10
11 // The original C code and the long comment below are
12 // from FreeBSD's /usr/src/lib/msun/src/s_erf.c and
13 // came with this notice. The go code is a simplified
14 // version of the original C.
15 //
16 // =====
17 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
18 //
19 // Developed at SunPro, a Sun Microsystems, Inc. business.
20 // Permission to use, copy, modify, and distribute this
21 // software is freely granted, provided that this notice
22 // is preserved.
23 // =====
24 //
25 //
26 // double erf(double x)
27 // double erfc(double x)
28 //
29 //          x
30 //          2   | \
31 // erf(x) = ----- | exp(-t*t)dt
32 //          sqrt(pi) \ |
33 //                   0
34 //
35 // erfc(x) = 1-erf(x)
36 // Note that
37 //          erf(-x) = -erf(x)
38 //          erfc(-x) = 2 - erfc(x)
39 //
40 // Method:
41 // 1. For |x| in [0, 0.84375]
42 //    erf(x) = x + x*R(x**2)
43 //    erfc(x) = 1 - erf(x) if x in [-.84375,
44 //    = 0.5 + ((0.5-x)-x*R) if x in [0.25,0.8
45 //    where R = P/Q where P is an odd poly of degree 8
```

```

45 //      Q is an odd poly of degree 10.
46 //
47 //      | R - (erf(x)-x)/x | <= 2-57.90
48 //
49 //
50 //      Remark. The formula is derived by noting
51 //      erf(x) = (2/sqrt(pi))*(x - x**3/3 + x**5/10 - x**
52 //      and that
53 //      2/sqrt(pi) = 1.128379167095512573896158903121545
54 //      is close to one. The interval is chosen because t
55 //      point of erf(x) is near 0.6174 (i.e., erf(x)=x wh
56 //      near 0.6174), and by some experiment, 0.84375 is
57 //      guarantee the error is less than one ulp for erf.
58 //
59 //      2. For |x| in [0.84375,1.25], let s = |x| - 1, and
60 //      c = 0.84506291151 rounded to single (24 bits)
61 //      erf(x) = sign(x) * (c + P1(s)/Q1(s))
62 //      erfc(x) = (1-c) - P1(s)/Q1(s) if x > 0
63 //      1+(c+P1(s)/Q1(s)) if x < 0
64 //      |P1/Q1 - (erf(|x|)-c)| <= 2**-59.06
65 //      Remark: here we use the taylor series expansion a
66 //      erf(1+s) = erf(1) + s*Poly(s)
67 //      = 0.845.. + P1(s)/Q1(s)
68 //      That is, we use rational approximation to approxi
69 //      erf(1+s) - (c = (single)0.8450629115
70 //      Note that |P1/Q1| < 0.078 for x in [0.84375,1.25]
71 //      where
72 //      P1(s) = degree 6 poly in s
73 //      Q1(s) = degree 6 poly in s
74 //
75 //      3. For x in [1.25,1/0.35(~2.857143)],
76 //      erfc(x) = (1/x)*exp(-x*x-0.5625+R1/S1)
77 //      erf(x) = 1 - erfc(x)
78 //      where
79 //      R1(z) = degree 7 poly in z, (z=1/x**2)
80 //      S1(z) = degree 8 poly in z
81 //
82 //      4. For x in [1/0.35,28]
83 //      erfc(x) = (1/x)*exp(-x*x-0.5625+R2/S2) if x
84 //      = 2.0 - (1/x)*exp(-x*x-0.5625+R2/S2)
85 //      = 2.0 - tiny (if x <= -6)
86 //      erf(x) = sign(x)*(1.0 - erfc(x)) if x < 6,
87 //      erf(x) = sign(x)*(1.0 - tiny)
88 //      where
89 //      R2(z) = degree 6 poly in z, (z=1/x**2)
90 //      S2(z) = degree 7 poly in z
91 //
92 //      Note1:
93 //      To compute exp(-x*x-0.5625+R/S), let s be a singl
94 //      precision number and s := x; then

```

```

95 //          -x*x = -s*s + (s-x)*(s+x)
96 //          exp(-x*x-0.5626+R/S) =
97 //          exp(-s*s-0.5625)*exp((s-x)*(s+x)+R/S)
98 //
99 // Note2:
100 //      Here 4 and 5 make use of the asymptotic series
101 //          exp(-x*x)
102 //          erfc(x) ~ ----- * ( 1 + Poly(1/x**2) )
103 //                      x*sqrt(pi)
104 //      We use rational approximation to approximate
105 //          g(s)=f(1/x**2) = log(erfc(x)*x) - x*x + 0.56
106 //      Here is the error bound for R1/S1 and R2/S2
107 //          |R1/S1 - f(x)| < 2**(-62.57)
108 //          |R2/S2 - f(x)| < 2**(-61.52)
109 //
110 // 5. For inf > x >= 28
111 //      erf(x) = sign(x) *(1 - tiny) (raise inexac
112 //      erfc(x) = tiny*tiny (raise underflow) if x >
113 //              = 2 - tiny if x<0
114 //
115 // 7. Special case:
116 //      erf(0) = 0, erf(inf) = 1, erf(-inf) = -1,
117 //      erfc(0) = 1, erfc(inf) = 0, erfc(-inf) = 2,
118 //      erfc/erf(NaN) is NaN
119 //
120 // const (
121 //      erx = 8.45062911510467529297e-01 // 0x3FEB0AC1600000
122 //      // Coefficients for approximation to erf in [0, 0.8
123 //      efx = 1.28379167095512586316e-01 // 0x3FC06EBA8214
124 //      efx8 = 1.02703333676410069053e+00 // 0x3FF06EBA8214
125 //      pp0 = 1.28379167095512558561e-01 // 0x3FC06EBA8214
126 //      pp1 = -3.25042107247001499370e-01 // 0xBFD4CD7D691C
127 //      pp2 = -2.84817495755985104766e-02 // 0xBF9D2A51DBD7
128 //      pp3 = -5.77027029648944159157e-03 // 0xBF77A2912366
129 //      pp4 = -2.37630166566501626084e-05 // 0xBEF8EAD61200
130 //      qq1 = 3.97917223959155352819e-01 // 0x3FD97779CDDA
131 //      qq2 = 6.50222499887672944485e-02 // 0x3FB0A54C5536
132 //      qq3 = 5.08130628187576562776e-03 // 0x3F74D022C4D3
133 //      qq4 = 1.32494738004321644526e-04 // 0x3F215DC9221C
134 //      qq5 = -3.96022827877536812320e-06 // 0xBED09C4342A2
135 //      // Coefficients for approximation to erf in [0.843
136 //      pa0 = -2.36211856075265944077e-03 // 0xBF6359B8BEF77
137 //      pa1 = 4.14856118683748331666e-01 // 0x3FDA8D00AD92E
138 //      pa2 = -3.72207876035701323847e-01 // 0xBFD7D240FBB8C
139 //      pa3 = 3.18346619901161753674e-01 // 0x3FD45FCA80512
140 //      pa4 = -1.10894694282396677476e-01 // 0xBFBC63983D3E2
141 //      pa5 = 3.54783043256182359371e-02 // 0x3FA22A3659979
142 //      pa6 = -2.16637559486879084300e-03 // 0xBF61BF380A960
143 //      qa1 = 1.06420880400844228286e-01 // 0x3FBB3E6618EEE
144 //      qa2 = 5.40397917702171048937e-01 // 0x3FE14AF092EB6

```

```

144     qa3 = 7.18286544141962662868e-02 // 0x3FB2635CD99FE
145     qa4 = 1.26171219808761642112e-01 // 0x3FC02660E7633
146     qa5 = 1.36370839120290507362e-02 // 0x3F8BEDC26B51D
147     qa6 = 1.19844998467991074170e-02 // 0x3F888B5457351
148     // Coefficients for approximation to erfc in [1.25,
149     ra0 = -9.86494403484714822705e-03 // 0xBF843412600D6
150     ra1 = -6.93858572707181764372e-01 // 0xBFE63416E4BA7
151     ra2 = -1.05586262253232909814e+01 // 0xC0251E0441B0E
152     ra3 = -6.23753324503260060396e+01 // 0xC04F300AE4CBA
153     ra4 = -1.62396669462573470355e+02 // 0xC0644CB184282
154     ra5 = -1.84605092906711035994e+02 // 0xC067135CEBCCA
155     ra6 = -8.12874355063065934246e+01 // 0xC054526557E4D
156     ra7 = -9.81432934416914548592e+00 // 0xC023A0EFC69AC
157     sa1 = 1.96512716674392571292e+01 // 0x4033A6B9BD707
158     sa2 = 1.37657754143519042600e+02 // 0x4061350C526AE
159     sa3 = 4.34565877475229228821e+02 // 0x407B290DD58A1
160     sa4 = 6.45387271733267880336e+02 // 0x40842B1921EC2
161     sa5 = 4.29008140027567833386e+02 // 0x407AD02157700
162     sa6 = 1.08635005541779435134e+02 // 0x405B28A3EE48A
163     sa7 = 6.57024977031928170135e+00 // 0x401A47EF8E484
164     sa8 = -6.04244152148580987438e-02 // 0xBF AEFF2EE749
165     // Coefficients for approximation to erfc in [1/.35
166     rb0 = -9.86494292470009928597e-03 // 0xBF84341239E86
167     rb1 = -7.99283237680523006574e-01 // 0xBFE993BA70C28
168     rb2 = -1.77579549177547519889e+01 // 0xC031C209555F9
169     rb3 = -1.60636384855821916062e+02 // 0xC064145D43C5E
170     rb4 = -6.37566443368389627722e+02 // 0xC083EC881375F
171     rb5 = -1.02509513161107724954e+03 // 0xC09004616A2E5
172     rb6 = -4.83519191608651397019e+02 // 0xC07E384E9BDC3
173     sb1 = 3.03380607434824582924e+01 // 0x403E568B261D5
174     sb2 = 3.25792512996573918826e+02 // 0x40745CAE221B9
175     sb3 = 1.53672958608443695994e+03 // 0x409802EB189D5
176     sb4 = 3.19985821950859553908e+03 // 0x40A8FFB7688C2
177     sb5 = 2.55305040643316442583e+03 // 0x40A3F219CEDF3
178     sb6 = 4.74528541206955367215e+02 // 0x407DA874E79FE
179     sb7 = -2.24409524465858183362e+01 // 0xC03670E242712
180 )
181
182 // Erf(x) returns the error function of x.
183 //
184 // Special cases are:
185 //     Erf(+Inf) = 1
186 //     Erf(-Inf) = -1
187 //     Erf(NaN) = NaN
188 func Erf(x float64) float64 {
189     const (
190         VeryTiny = 2.848094538889218e-306 // 0x00800
191         Small    = 1.0 / (1 << 28)      // 2** -28
192     )

```

```

193 // special cases
194 switch {
195 case IsNaN(x):
196     return NaN()
197 case IsInf(x, 1):
198     return 1
199 case IsInf(x, -1):
200     return -1
201 }
202 sign := false
203 if x < 0 {
204     x = -x
205     sign = true
206 }
207 if x < 0.84375 { // |x| < 0.84375
208     var temp float64
209     if x < Small { // |x| < 2**-28
210         if x < VeryTiny {
211             temp = 0.125 * (8.0*x + efx8)
212         } else {
213             temp = x + efx*x
214         }
215     } else {
216         z := x * x
217         r := pp0 + z*(pp1+z*(pp2+z*(pp3+z*pp
218         s := 1 + z*(qq1+z*(qq2+z*(qq3+z*(qq4
219         y := r / s
220         temp = x + x*y
221     }
222     if sign {
223         return -temp
224     }
225     return temp
226 }
227 if x < 1.25 { // 0.84375 <= |x| < 1.25
228     s := x - 1
229     P := pa0 + s*(pa1+s*(pa2+s*(pa3+s*(pa4+s*(pa
230     Q := 1 + s*(qa1+s*(qa2+s*(qa3+s*(qa4+s*(qa5+
231     if sign {
232         return -erx - P/Q
233     }
234     return erx + P/Q
235 }
236 if x >= 6 { // inf > |x| >= 6
237     if sign {
238         return -1
239     }
240     return 1
241 }
242 s := 1 / (x * x)

```

```

243     var R, S float64
244     if x < 1/0.35 { // |x| < 1 / 0.35 ~ 2.857143
245         R = ra0 + s*(ra1+s*(ra2+s*(ra3+s*(ra4+s*(ra5
246         S = 1 + s*(sa1+s*(sa2+s*(sa3+s*(sa4+s*(sa5+s
247     } else { // |x| >= 1 / 0.35 ~ 2.857143
248         R = rb0 + s*(rb1+s*(rb2+s*(rb3+s*(rb4+s*(rb5
249         S = 1 + s*(sb1+s*(sb2+s*(sb3+s*(sb4+s*(sb5+s
250     }
251     z := Float64frombits(Float64bits(x) & 0xffffffff0000
252     r := Exp(-z*z-0.5625) * Exp((z-x)*(z+x)+R/S)
253     if sign {
254         return r/x - 1
255     }
256     return 1 - r/x
257 }
258
259 // Erfc(x) returns the complementary error function of x.
260 //
261 // Special cases are:
262 //     Erfc(+Inf) = 0
263 //     Erfc(-Inf) = 2
264 //     Erfc(NaN) = NaN
265 func Erfc(x float64) float64 {
266     const Tiny = 1.0 / (1 << 56) // 2**-56
267     // special cases
268     switch {
269     case IsNaN(x):
270         return NaN()
271     case IsInf(x, 1):
272         return 0
273     case IsInf(x, -1):
274         return 2
275     }
276     sign := false
277     if x < 0 {
278         x = -x
279         sign = true
280     }
281     if x < 0.84375 { // |x| < 0.84375
282         var temp float64
283         if x < Tiny { // |x| < 2**-56
284             temp = x
285         } else {
286             z := x * x
287             r := pp0 + z*(pp1+z*(pp2+z*(pp3+z*pp
288             s := 1 + z*(qq1+z*(qq2+z*(qq3+z*(qq4
289             y := r / s
290             if x < 0.25 { // |x| < 1/4
291                 temp = x + x*y

```

```

292             } else {
293                 temp = 0.5 + (x*y + (x - 0.5
294             }
295         }
296         if sign {
297             return 1 + temp
298         }
299         return 1 - temp
300     }
301     if x < 1.25 { // 0.84375 <= |x| < 1.25
302         s := x - 1
303         P := pa0 + s*(pa1+s*(pa2+s*(pa3+s*(pa4+s*(pa
304         Q := 1 + s*(qa1+s*(qa2+s*(qa3+s*(qa4+s*(qa5+
305         if sign {
306             return 1 + erx + P/Q
307         }
308         return 1 - erx - P/Q
309     }
310 }
311 if x < 28 { // |x| < 28
312     s := 1 / (x * x)
313     var R, S float64
314     if x < 1/0.35 { // |x| < 1 / 0.35 ~ 2.857143
315         R = ra0 + s*(ra1+s*(ra2+s*(ra3+s*(ra
316         S = 1 + s*(sa1+s*(sa2+s*(sa3+s*(sa4+
317     } else { // |x| >= 1 / 0.35 ~ 2.857143
318         if sign && x > 6 {
319             return 2 // x < -6
320         }
321         R = rb0 + s*(rb1+s*(rb2+s*(rb3+s*(rb
322         S = 1 + s*(sb1+s*(sb2+s*(sb3+s*(sb4+
323     }
324     z := Float64frombits(Float64bits(x) & 0xffff
325     r := Exp(-z*z-0.5625) * Exp((z-x)*(z+x)+R/S)
326     if sign {
327         return 2 - r/x
328     }
329     return r / x
330 }
331 if sign {
332     return 2
333 }
334 return 0
335 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/exp.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Exp returns e**x, the base-e exponential of x.
8 //
9 // Special cases are:
10 //     Exp(+Inf) = +Inf
11 //     Exp(NaN) = NaN
12 // Very large values overflow to 0 or +Inf.
13 // Very small values underflow to 1.
14 func Exp(x float64) float64
15
16 // The original C code, the long comment, and the constants
17 // below are from FreeBSD's /usr/src/lib/msun/src/e_exp.c
18 // and came with this notice. The go code is a simplified
19 // version of the original C.
20 //
21 // =====
22 // Copyright (C) 2004 by Sun Microsystems, Inc. All rights r
23 //
24 // Permission to use, copy, modify, and distribute this
25 // software is freely granted, provided that this notice
26 // is preserved.
27 // =====
28 //
29 //
30 // exp(x)
31 // Returns the exponential of x.
32 //
33 // Method
34 // 1. Argument reduction:
35 //     Reduce x to an r so that |r| <= 0.5*ln2 ~ 0.34658.
36 //     Given x, find r and integer k such that
37 //
38 //         x = k*ln2 + r, |r| <= 0.5*ln2.
39 //
40 //     Here r will be represented as r = hi-lo for better
41 //     accuracy.
42 //
43 // 2. Approximation of exp(r) by a special rational functi
44 // the interval [0,0.34658]:
```

```

45 // Write
46 //     R(r**2) = r*(exp(r)+1)/(exp(r)-1) = 2 + r*r/6 -
47 // We use a special Remes algorithm on [0,0.34658] to g
48 // a polynomial of degree 5 to approximate R. The maxim
49 // of this polynomial approximation is bounded by 2**-5
50 // other words,
51 //     R(z) ~ 2.0 + P1*z + P2*z**2 + P3*z**3 + P4*z**4
52 // (where z=r*r, and the values of P1 to P5 are listed
53 // and
54 //     | 2.0+P1*z+...+P5*z5 - R(z) | <= 2-59
55 //     |
56 // The computation of exp(r) thus becomes
57 //
58 //     2*r
59 //     exp(r) = 1 + -----
60 //                   R - r
61 //                   r*R1(r)
62 //     = 1 + r + ----- (for better acc
63 //                   2 - R1(r)
64 // where
65 //
66 //     2      4      10
67 //     R1(r) = r - (P1*r  + P2*r  + ... + P5*r  ).
68 // 3. Scale back to obtain exp(x):
69 // From step 1, we have
70 //     exp(x) = 2**k * exp(r)
71 //
72 // Special cases:
73 //     exp(INF) is INF, exp(NaN) is NaN;
74 //     exp(-INF) is 0, and
75 //     for finite argument, only exp(0)=1 is exact.
76 //
77 // Accuracy:
78 //     according to an error analysis, the error is always
79 //     1 ulp (unit in the last place).
80 //
81 // Misc. info.
82 //     For IEEE double
83 //         if x > 7.09782712893383973096e+02 then exp(x) o
84 //         if x < -7.45133219101941108420e+02 then exp(x) u
85 //
86 // Constants:
87 // The hexadecimal values are the intended ones for the foll
88 // constants. The decimal values may be used, provided that
89 // compiler will convert from decimal to binary accurately e
90 // to produce the hexadecimal values shown.
91
92 func exp(x float64) float64 {
93     const (
94         Ln2Hi = 6.93147180369123816490e-01

```

```

95             Ln2Lo = 1.90821492927058770002e-10
96             Log2e = 1.44269504088896338700e+00
97
98             Overflow = 7.09782712893383973096e+02
99             Underflow = -7.45133219101941108420e+02
100            NearZero = 1.0 / (1 << 28) // 2** -28
101        )
102
103        // special cases
104        switch {
105        case IsNaN(x) || IsInf(x, 1):
106            return x
107        case IsInf(x, -1):
108            return 0
109        case x > Overflow:
110            return Inf(1)
111        case x < Underflow:
112            return 0
113        case -NearZero < x && x < NearZero:
114            return 1 + x
115        }
116
117        // reduce; computed as r = hi - lo for extra precisi
118        var k int
119        switch {
120        case x < 0:
121            k = int(Log2e*x - 0.5)
122        case x > 0:
123            k = int(Log2e*x + 0.5)
124        }
125        hi := x - float64(k)*Ln2Hi
126        lo := float64(k) * Ln2Lo
127
128        // compute
129        return expmulti(hi, lo, k)
130    }
131
132    // Exp2 returns 2**x, the base-2 exponential of x.
133    //
134    // Special cases are the same as Exp.
135    func Exp2(x float64) float64
136
137    func exp2(x float64) float64 {
138        const (
139            Ln2Hi = 6.93147180369123816490e-01
140            Ln2Lo = 1.90821492927058770002e-10
141
142            Overflow = 1.0239999999999999e+03
143            Underflow = -1.0740e+03

```

```

144     )
145
146     // special cases
147     switch {
148     case IsNaN(x) || IsInf(x, 1):
149         return x
150     case IsInf(x, -1):
151         return 0
152     case x > Overflow:
153         return Inf(1)
154     case x < Underflow:
155         return 0
156     }
157
158     // argument reduction; x = r*lg(e) + k with |r| ≤ ln
159     // computed as r = hi - lo for extra precision.
160     var k int
161     switch {
162     case x > 0:
163         k = int(x + 0.5)
164     case x < 0:
165         k = int(x - 0.5)
166     }
167     t := x - float64(k)
168     hi := t * Ln2Hi
169     lo := -t * Ln2Lo
170
171     // compute
172     return expmulti(hi, lo, k)
173 }
174
175 // exp1 returns e**r × 2**k where r = hi - lo and |r| ≤ ln(2)
176 func expmulti(hi, lo float64, k int) float64 {
177     const (
178         P1 = 1.666666666666666019037e-01 /* 0x3FC555
179         P2 = -2.777777777770155933842e-03 /* 0xBF66C1
180         P3 = 6.61375632143793436117e-05 /* 0x3F1156
181         P4 = -1.65339022054652515390e-06 /* 0xBEBBBD
182         P5 = 4.13813679705723846039e-08 /* 0x3E6637
183     )
184
185     r := hi - lo
186     t := r * r
187     c := r - t*(P1+t*(P2+t*(P3+t*(P4+t*P5))))
188     y := 1 - ((lo - (r*c)/(2-c)) - hi)
189     // TODO(rsc): make sure Ldexp can handle boundary k
190     return Ldexp(y, k)
191 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/expm1.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // The original C code, the long comment, and the constants
8 // below are from FreeBSD's /usr/src/lib/msun/src/s_expm1.c
9 // and came with this notice. The go code is a simplified
10 // version of the original C.
11 //
12 // =====
13 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
14 //
15 // Developed at SunPro, a Sun Microsystems, Inc. business.
16 // Permission to use, copy, modify, and distribute this
17 // software is freely granted, provided that this notice
18 // is preserved.
19 // =====
20 //
21 // expm1(x)
22 // Returns exp(x)-1, the exponential of x minus 1.
23 //
24 // Method
25 //   1. Argument reduction:
26 //      Given x, find r and integer k such that
27 //
28 //           $x = k \cdot \ln 2 + r, \quad |r| \leq 0.5 \cdot \ln 2 \sim 0.34658$ 
29 //
30 //      Here a correction term c will be computed to compensate
31 //      the error in r when rounded to a floating-point number.
32 //
33 //   2. Approximating expm1(r) by a special rational function on
34 //      the interval [0,0.34658]:
35 //      Since
36 //           $r \cdot (\exp(r)+1) / (\exp(r)-1) = 2 + r^2/6 - r^4/360 + \dots$ 
37 //      we define R1(r*r) by
38 //           $r \cdot (\exp(r)+1) / (\exp(r)-1) = 2 + r^2/6 * R1(r*r)$ 
39 //      That is,
40 //           $R1(r^2) = 6/r * ((\exp(r)+1)/(\exp(r)-1) - 2/r)$ 
41 //                   $= 6/r * (1 + 2.0*(1/(\exp(r)-1) - 1/r))$ 
42 //                   $= 1 - r^2/60 + r^4/2520 - r^6/100800 + \dots$ 
43 //      We use a special Reme algorithm on [0,0.347] to generate
44 //      a polynomial of degree 5 in r*r to approximate R1. T
```



```

95 //
96 // Special cases:
97 //     expm1(INF) is INF, expm1(NaN) is NaN;
98 //     expm1(-INF) is -1, and
99 //     for finite argument, only expm1(0)=0 is exact.
100 //
101 // Accuracy:
102 //     according to an error analysis, the error is always
103 //     1 ulp (unit in the last place).
104 //
105 // Misc. info.
106 //     For IEEE double
107 //         if x > 7.09782712893383973096e+02 then expm1(x)
108 //
109 // Constants:
110 // The hexadecimal values are the intended ones for the foll
111 // constants. The decimal values may be used, provided that
112 // compiler will convert from decimal to binary accurately e
113 // to produce the hexadecimal values shown.
114 //
115
116 // Expm1 returns e**x - 1, the base-e exponential of x minus
117 // It is more accurate than Exp(x) - 1 when x is near zero.
118 //
119 // Special cases are:
120 //     Expm1(+Inf) = +Inf
121 //     Expm1(-Inf) = -1
122 //     Expm1(NaN) = NaN
123 // Very large values overflow to -1 or +Inf.
124 func Expm1(x float64) float64
125
126 func expm1(x float64) float64 {
127     const (
128         Othreshold = 7.09782712893383973096e+02 // 0
129         Ln2X56     = 3.88162421113569373274e+01 // 0
130         Ln2HalfX3  = 1.03972077083991796413e+00 // 0
131         Ln2Half    = 3.46573590279972654709e-01 // 0
132         Ln2Hi      = 6.93147180369123816490e-01 // 0
133         Ln2Lo      = 1.90821492927058770002e-10 // 0
134         InvLn2     = 1.44269504088896338700e+00 // 0
135         Tiny       = 1.0 / (1 << 54)           // 2
136         // scaled coefficients related to expm1
137         Q1 = -3.333333333333331316428e-02 // 0xBFA111
138         Q2 = 1.58730158725481460165e-03  // 0x3F5A01
139         Q3 = -7.93650757867487942473e-05 // 0xBF14CE
140         Q4 = 4.00821782732936239552e-06  // 0x3ED0CF
141         Q5 = -2.01099218183624371326e-07 // 0xBE8AFD
142     )
143

```

```

144     // special cases
145     switch {
146     case IsInf(x, 1) || IsNaN(x):
147         return x
148     case IsInf(x, -1):
149         return -1
150     }
151
152     absx := x
153     sign := false
154     if x < 0 {
155         absx = -absx
156         sign = true
157     }
158
159     // filter out huge argument
160     if absx >= Ln2X56 { // if |x| >= 56 * ln2
161         if absx >= Othreshold { // if |x| >= 709.78.
162             return Inf(1) // overflow
163         }
164         if sign {
165             return -1 // x < -56*ln2, return -1.
166         }
167     }
168
169     // argument reduction
170     var c float64
171     var k int
172     if absx > Ln2Half { // if |x| > 0.5 * ln2
173         var hi, lo float64
174         if absx < Ln2HalfX3 { // and |x| < 1.5 * ln2
175             if !sign {
176                 hi = x - Ln2Hi
177                 lo = Ln2Lo
178                 k = 1
179             } else {
180                 hi = x + Ln2Hi
181                 lo = -Ln2Lo
182                 k = -1
183             }
184         } else {
185             if !sign {
186                 k = int(InvLn2*x + 0.5)
187             } else {
188                 k = int(InvLn2*x - 0.5)
189             }
190             t := float64(k)
191             hi = x - t*Ln2Hi // t * Ln2Hi is exa
192             lo = t * Ln2Lo

```

```

193     }
194     x = hi - lo
195     c = (hi - x) - lo
196 } else if absx < Tiny { // when |x| < 2** -54, return
197     return x
198 } else {
199     k = 0
200 }
201
202 // x is now in primary range
203 hfx := 0.5 * x
204 hxs := x * hfx
205 r1 := 1 + hxs*(Q1+hxs*(Q2+hxs*(Q3+hxs*(Q4+hxs*Q5)))
206 t := 3 - r1*hfx
207 e := hxs * ((r1 - t) / (6.0 - x*t))
208 if k != 0 {
209     e = (x*(e-c) - c)
210     e -= hxs
211     switch {
212     case k == -1:
213         return 0.5*(x-e) - 0.5
214     case k == 1:
215         if x < -0.25 {
216             return -2 * (e - (x + 0.5))
217         }
218         return 1 + 2*(x-e)
219     case k <= -2 || k > 56: // suffice to return
220         y := 1 - (e - x)
221         y = Float64frombits(Float64bits(y) +
222             return y - 1
223     }
224     if k < 20 {
225         t := Float64frombits(0x3ff0000000000000)
226         y := t - (e - x)
227         y = Float64frombits(Float64bits(y) +
228             return y
229     }
230     t := Float64frombits(uint64((0x3ff - k) << 5)
231     y := x - (e + t)
232     y += 1
233     y = Float64frombits(Float64bits(y) + uint64(
234     return y
235 }
236 return x - (x*e - hxs) // c is 0
237 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/floor.go

```
1 // Copyright 2009-2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Floor returns the greatest integer value less than or equ
8 //
9 // Special cases are:
10 //     Floor( $\pm 0$ ) =  $\pm 0$ 
11 //     Floor( $\pm \text{Inf}$ ) =  $\pm \text{Inf}$ 
12 //     Floor(NaN) = NaN
13 func Floor(x float64) float64
14
15 func floor(x float64) float64 {
16     if x == 0 || IsNaN(x) || IsInf(x, 0) {
17         return x
18     }
19     if x < 0 {
20         d, fract := Modf(-x)
21         if fract != 0.0 {
22             d = d + 1
23         }
24         return -d
25     }
26     d, _ := Modf(x)
27     return d
28 }
29
30 // Ceil returns the least integer value greater than or equ
31 //
32 // Special cases are:
33 //     Ceil( $\pm 0$ ) =  $\pm 0$ 
34 //     Ceil( $\pm \text{Inf}$ ) =  $\pm \text{Inf}$ 
35 //     Ceil(NaN) = NaN
36 func Ceil(x float64) float64
37
38 func ceil(x float64) float64 {
39     return -Floor(-x)
40 }
41
42 // Trunc returns the integer value of x.
43 //
44 // Special cases are:
```

```
45 //      Trunc( $\pm 0$ ) =  $\pm 0$ 
46 //      Trunc( $\pm \text{Inf}$ ) =  $\pm \text{Inf}$ 
47 //      Trunc(NaN) = NaN
48 func Trunc(x float64) float64
49
50 func trunc(x float64) float64 {
51     if x == 0 || IsNaN(x) || IsInf(x, 0) {
52         return x
53     }
54     d, _ := Modf(x)
55     return d
56 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/frexp.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Frexp breaks f into a normalized fraction
8 // and an integral power of two.
9 // It returns frac and exp satisfying f == frac × 2**exp,
10 // with the absolute value of frac in the interval [½, 1).
11 //
12 // Special cases are:
13 //     Frexp(±0) = ±0, 0
14 //     Frexp(±Inf) = ±Inf, 0
15 //     Frexp(NaN) = NaN, 0
16 func Frexp(f float64) (frac float64, exp int)
17
18 func frexp(f float64) (frac float64, exp int) {
19     // special cases
20     switch {
21     case f == 0:
22         return f, 0 // correctly return -0
23     case IsInf(f, 0) || IsNaN(f):
24         return f, 0
25     }
26     f, exp = normalize(f)
27     x := Float64bits(f)
28     exp += int((x>>shift)&mask) - bias + 1
29     x ^= mask << shift
30     x |= (-1 + bias) << shift
31     frac = Float64frombits(x)
32     return
33 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/gamma.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // The original C code, the long comment, and the constants
8 // below are from http://netlib.sandia.gov/cephes/cprob/gamr
9 // The go code is a simplified version of the original C.
10 //
11 //      tgamma.c
12 //
13 //      Gamma function
14 //
15 // SYNOPSIS:
16 //
17 // double x, y, tgamma();
18 // extern int signgam;
19 //
20 // y = tgamma( x );
21 //
22 // DESCRIPTION:
23 //
24 // Returns gamma function of the argument. The result is
25 // correctly signed, and the sign (+1 or -1) is also
26 // returned in a global (extern) variable named signgam.
27 // This variable is also filled in by the logarithmic gamma
28 // function lgamma().
29 //
30 // Arguments |x| <= 34 are reduced by recurrence and the fun
31 // approximated by a rational function of degree 6/7 in the
32 // interval (2,3). Large arguments are handled by Stirling'
33 // formula. Large negative arguments are made positive using
34 // a reflection formula.
35 //
36 // ACCURACY:
37 //
38 //                                     Relative error:
39 // arithmetic      domain      # trials      peak      rms
40 //      DEC         -34, 34      10000        1.3e-16    2.5e-17
41 //      IEEE        -170, -33    20000        2.3e-15    3.3e-16
42 //      IEEE         -33, 33     20000        9.4e-16    2.2e-16
43 //      IEEE          33, 171.6  20000        2.3e-15    3.2e-16
44 //
```

```

45 // Error for arguments outside the test range will be larger
46 // owing to error amplification by the exponential function.
47 //
48 // Cephes Math Library Release 2.8:  June, 2000
49 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
50 //
51 // The readme file at http://netlib.sandia.gov/cephes/ says:
52 //   Some software in this archive may be from the book _Me
53 // Programs for Mathematical Functions_ (Prentice-Hall or Si
54 // International, 1989) or from the Cephes Mathematical Libr
55 // commercial product. In either event, it is copyrighted by
56 // What you see here may be used freely but it comes with no
57 // guarantee.
58 //
59 //   The two known misprints in the book are repaired here i
60 // source listings for the gamma function and the incomplete
61 // integral.
62 //
63 //   Stephen L. Moshier
64 //   moshier@na-net.ornl.gov
65
66 var _gamP = [...]float64{
67     1.60119522476751861407e-04,
68     1.19135147006586384913e-03,
69     1.04213797561761569935e-02,
70     4.76367800457137231464e-02,
71     2.07448227648435975150e-01,
72     4.94214826801497100753e-01,
73     9.9999999999999996796e-01,
74 }
75 var _gamQ = [...]float64{
76     -2.31581873324120129819e-05,
77     5.39605580493303397842e-04,
78     -4.45641913851797240494e-03,
79     1.18139785222060435552e-02,
80     3.58236398605498653373e-02,
81     -2.34591795718243348568e-01,
82     7.14304917030273074085e-02,
83     1.00000000000000000320e+00,
84 }
85 var _gamS = [...]float64{
86     7.87311395793093628397e-04,
87     -2.29549961613378126380e-04,
88     -2.68132617805781232825e-03,
89     3.47222221605458667310e-03,
90     8.33333333333482257126e-02,
91 }
92
93 // Gamma function computed by Stirling's formula.
94 // The polynomial is valid for 33 <= x <= 172.

```

```

95 func stirling(x float64) float64 {
96     const (
97         SqrtTwoPi    = 2.506628274631000502417
98         MaxStirling  = 143.01608
99     )
100    w := 1 / x
101    w = 1 + w*(((gamS[0]*w+_gamS[1])*w+_gamS[2])*w+_ga
102    y := Exp(x)
103    if x > MaxStirling { // avoid Pow() overflow
104        v := Pow(x, 0.5*x-0.25)
105        y = v * (v / y)
106    } else {
107        y = Pow(x, x-0.5) / y
108    }
109    y = SqrtTwoPi * y * w
110    return y
111 }
112
113 // Gamma(x) returns the Gamma function of x.
114 //
115 // Special cases are:
116 //     Gamma( $\pm$ Inf) =  $\pm$ Inf
117 //     Gamma(NaN) = NaN
118 // Large values overflow to +Inf.
119 // Zero and negative integer arguments return  $\pm$ Inf.
120 func Gamma(x float64) float64 {
121     const Euler = 0.577215664901532860606512090082402431
122     // special cases
123     switch {
124     case IsInf(x, -1) || IsNaN(x):
125         return x
126     case x < -170.5674972726612 || x > 171.6144788718229
127         return Inf(1)
128     }
129     q := Abs(x)
130     p := Floor(q)
131     if q > 33 {
132         if x >= 0 {
133             return stirling(x)
134         }
135         signgam := 1
136         if ip := int(p); ip&1 == 0 {
137             signgam = -1
138         }
139         z := q - p
140         if z > 0.5 {
141             p = p + 1
142             z = q - p
143         }

```

```

144         z = q * Sin(Pi*z)
145         if z == 0 {
146             return Inf(singgam)
147         }
148         z = Pi / (Abs(z) * stirling(q))
149         return float64(singgam) * z
150     }
151
152     // Reduce argument
153     z := 1.0
154     for x >= 3 {
155         x = x - 1
156         z = z * x
157     }
158     for x < 0 {
159         if x > -1e-09 {
160             goto small
161         }
162         z = z / x
163         x = x + 1
164     }
165     for x < 2 {
166         if x < 1e-09 {
167             goto small
168         }
169         z = z / x
170         x = x + 1
171     }
172
173     if x == 2 {
174         return z
175     }
176
177     x = x - 2
178     p = (((((x*_gamP[0]+_gamP[1])*x+_gamP[2])*x+_gamP[3]
179     q = (((((x*_gamQ[0]+_gamQ[1])*x+_gamQ[2])*x+_gamQ[3]
180     return z * p / q
181
182     small:
183         if x == 0 {
184             return Inf(1)
185         }
186         return z / ((1 + Euler*x) * x)
187     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/hypot.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Hypot -- sqrt(p*p + q*q), but overflows only if the
9 */
10
11 // Hypot computes Sqrt(p*p + q*q), taking care to avoid
12 // unnecessary overflow and underflow.
13 //
14 // Special cases are:
15 //     Hypot(p, q) = +Inf if p or q is infinite
16 //     Hypot(p, q) = NaN if p or q is NaN
17 func Hypot(p, q float64) float64
18
19 func hypot(p, q float64) float64 {
20     // special cases
21     switch {
22     case IsInf(p, 0) || IsInf(q, 0):
23         return Inf(1)
24     case IsNaN(p) || IsNaN(q):
25         return NaN()
26     }
27     if p < 0 {
28         p = -p
29     }
30     if q < 0 {
31         q = -q
32     }
33     if p < q {
34         p, q = q, p
35     }
36     if p == 0 {
37         return 0
38     }
39     q = q / p
40     return p * Sqrt(1+q*q)
41 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/j0.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8      Bessel function of the first and second kinds of order
9 */
10
11 // The original C code and the long comment below are
12 // from FreeBSD's /usr/src/lib/msun/src/e_j0.c and
13 // came with this notice. The go code is a simplified
14 // version of the original C.
15 //
16 // =====
17 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
18 //
19 // Developed at SunPro, a Sun Microsystems, Inc. business.
20 // Permission to use, copy, modify, and distribute this
21 // software is freely granted, provided that this notice
22 // is preserved.
23 // =====
24 //
25 // __ieee754_j0(x), __ieee754_y0(x)
26 // Bessel function of the first and second kinds of order zero
27 // Method -- j0(x):
28 //     1. For tiny x, we use  $j_0(x) = 1 - x^2/4 + x^4/64 -$ 
29 //     2. Reduce x to |x| since  $j_0(x)=j_0(-x)$ , and
30 //     for x in (0,2)
31 //          $j_0(x) = 1 - z/4 + z^2 R_0/S_0$ , where  $z = x*x$ ;
32 //     (precision:  $|j_0 - 1 + z/4 - z^2 R_0/S_0| < 2^{-63.67}$ )
33 //     for x in (2,inf)
34 //          $j_0(x) = \sqrt{2/(pi*x)} * (p_0(x)*\cos(x_0) - q_0(x))$ 
35 //     where  $x_0 = x - pi/4$ . It is better to compute  $\sin(x_0)$ 
36 //     as follow:
37 //          $\cos(x_0) = \cos(x)\cos(pi/4) + \sin(x)\sin(pi/4)$ 
38 //          $= 1/\sqrt{2} * (\cos(x) + \sin(x))$ 
39 //          $\sin(x_0) = \sin(x)\cos(pi/4) - \cos(x)\sin(pi/4)$ 
40 //          $= 1/\sqrt{2} * (\sin(x) - \cos(x))$ 
41 //     (To avoid cancellation, use
42 //          $\sin(x) + \cos(x) = -\cos(2x)/(\sin(x) - \cos(x))$ 
43 //     to compute the worse one.)
44 //
```

```

45 //      3 Special cases
46 //          j0(nan)= nan
47 //          j0(0) = 1
48 //          j0(inf) = 0
49 //
50 // Method -- y0(x):
51 //      1. For x<2.
52 //          Since
53 //               $y_0(x) = 2/\pi * (j_0(x) * (\ln(x/2) + \text{Euler}) + x^{2/4})$ 
54 //          therefore  $y_0(x) - 2/\pi * j_0(x) * \ln(x)$  is an even funct
55 //          We use the following function to approximate  $y_0$ ,
56 //               $y_0(x) = U(z)/V(z) + (2/\pi) * (j_0(x) * \ln(x))$ ,  $z =$ 
57 //          where
58 //               $U(z) = u_{00} + u_{01} * z + \dots + u_{06} * z^{**6}$ 
59 //               $V(z) = 1 + v_{01} * z + \dots + v_{04} * z^{**4}$ 
60 //          with absolute approximation error bounded by  $2^{**-}$ 
61 //          Note: For tiny x,  $U/V = u_0$  and  $j_0(x) \sim 1$ , hence
62 //               $y_0(\text{tiny}) = u_0 + (2/\pi) * \ln(\text{tiny})$ , (choose tin
63 //      2. For  $x \geq 2$ .
64 //               $y_0(x) = \text{sqrt}(2/(\pi * x)) * (p_0(x) * \cos(x_0) + q_0(x) *$ 
65 //          where  $x_0 = x - \pi/4$ . It is better to compute  $\sin(x_0)$ 
66 //          by the method mentioned above.
67 //      3. Special cases:  $y_0(0) = -\text{inf}$ ,  $y_0(x < 0) = \text{NaN}$ ,  $y_0(\text{inf}) = 0$ 
68 //
69
70 // J0 returns the order-zero Bessel function of the first ki
71 //
72 // Special cases are:
73 //      J0( $\pm \text{Inf}$ ) = 0
74 //      J0(0) = 1
75 //      J0(NaN) = NaN
76 func J0(x float64) float64 {
77     const (
78         Huge      = 1e300
79         TwoM27    = 1.0 / (1 << 27) // 2** -27 0x3e40000
80         TwoM13    = 1.0 / (1 << 13) // 2** -13 0x3f20000
81         Two129    = 1 << 129        // 2** 129 0x4800000
82         // R0/S0 on [0, 2]
83         R02      = 1.56249999999999947958e-02 // 0x3F8FF
84         R03      = -1.89979294238854721751e-04 // 0xBF28E
85         R04      = 1.82954049532700665670e-06 // 0x3EBEB
86         R05      = -4.61832688532103189199e-09 // 0xBE33D
87         S01      = 1.56191029464890010492e-02 // 0x3F8FF
88         S02      = 1.16926784663337450260e-04 // 0x3F1EA
89         S03      = 5.13546550207318111446e-07 // 0x3EA13
90         S04      = 1.16614003333790000205e-09 // 0x3E140
91     )
92     // special cases
93     switch {
94     case IsNaN(x):

```

```

95         return x
96     case IsInf(x, 0):
97         return 0
98     case x == 0:
99         return 1
100    }
101
102    if x < 0 {
103        x = -x
104    }
105    if x >= 2 {
106        s, c := Sincos(x)
107        ss := s - c
108        cc := s + c
109
110        // make sure x+x does not overflow
111        if x < MaxFloat64/2 {
112            z := -Cos(x + x)
113            if s*c < 0 {
114                cc = z / ss
115            } else {
116                ss = z / cc
117            }
118        }
119
120        // j0(x) = 1/sqrt(pi) * (P(0,x)*cc - Q(0,x)*
121        // y0(x) = 1/sqrt(pi) * (P(0,x)*ss + Q(0,x)*
122
123        var z float64
124        if x > Two129 { // |x| > ~6.8056e+38
125            z = (1 / SqrtPi) * cc / Sqrt(x)
126        } else {
127            u := pzero(x)
128            v := qzero(x)
129            z = (1 / SqrtPi) * (u*cc - v*ss) / s
130        }
131        return z // |x| >= 2.0
132    }
133    if x < TwoM13 { // |x| < ~1.2207e-4
134        if x < TwoM27 {
135            return 1 // |x| < ~7.4506e-9
136        }
137        return 1 - 0.25*x*x // ~7.4506e-9 < |x| < ~1
138    }
139    z := x * x
140    r := z * (R02 + z*(R03+z*(R04+z*R05)))
141    s := 1 + z*(S01+z*(S02+z*(S03+z*S04)))
142    if x < 1 {
143        return 1 + z*(-0.25+(r/s)) // |x| < 1.00

```

```

144     }
145     u := 0.5 * x
146     return (1+u)*(1-u) + z*(r/s) // 1.0 < |x| < 2.0
147 }
148
149 // Y0 returns the order-zero Bessel function of the second k
150 //
151 // Special cases are:
152 //     Y0(+Inf) = 0
153 //     Y0(0) = -Inf
154 //     Y0(x < 0) = NaN
155 //     Y0(NaN) = NaN
156 func Y0(x float64) float64 {
157     const (
158         TwoM27 = 1.0 / (1 << 27)           // 2**-
159         Two129 = 1 << 129                 // 2**1
160         U00    = -7.38042951086872317523e-02 // 0xBF
161         U01    = 1.76666452509181115538e-01  // 0x3F
162         U02    = -1.38185671945596898896e-02 // 0xBF
163         U03    = 3.47453432093683650238e-04  // 0x3F
164         U04    = -3.81407053724364161125e-06 // 0xBE
165         U05    = 1.95590137035022920206e-08  // 0x3E
166         U06    = -3.98205194132103398453e-11 // 0xBD
167         V01    = 1.27304834834123699328e-02  // 0x3F
168         V02    = 7.60068627350353253702e-05  // 0x3F
169         V03    = 2.59150851840457805467e-07  // 0x3E
170         V04    = 4.41110311332675467403e-10  // 0x3D
171     )
172     // special cases
173     switch {
174     case x < 0 || IsNaN(x):
175         return NaN()
176     case IsInf(x, 1):
177         return 0
178     case x == 0:
179         return Inf(-1)
180     }
181
182     if x >= 2 { // |x| >= 2.0
183
184         // y0(x) = sqrt(2/(pi*x))*(p0(x)*sin(x0)+q0(
185         //     where x0 = x-pi/4
186         // Better formula:
187         //     cos(x0) = cos(x)cos(pi/4)+sin(x)sin(p
188         //             = 1/sqrt(2) * (sin(x) + cos(
189         //     sin(x0) = sin(x)cos(3pi/4)-cos(x)sin(
190         //             = 1/sqrt(2) * (sin(x) - cos(
191         // To avoid cancellation, use
192         //     sin(x) +- cos(x) = -cos(2x)/(sin(x) -

```

```

193         // to compute the worse one.
194
195         s, c := Sincos(x)
196         ss := s - c
197         cc := s + c
198
199         //  $j_0(x) = 1/\sqrt{\pi} * (P(0,x)*cc - Q(0,x)*$ 
200         //  $y_0(x) = 1/\sqrt{\pi} * (P(0,x)*ss + Q(0,x)*$ 
201
202         // make sure x+x does not overflow
203         if x < MaxFloat64/2 {
204             z := -Cos(x + x)
205             if s*c < 0 {
206                 cc = z / ss
207             } else {
208                 ss = z / cc
209             }
210         }
211         var z float64
212         if x > Two129 { // |x| > ~6.8056e+38
213             z = (1 / SqrtPi) * ss / Sqrt(x)
214         } else {
215             u := pzero(x)
216             v := qzero(x)
217             z = (1 / SqrtPi) * (u*ss + v*cc) / S
218         }
219         return z // |x| >= 2.0
220     }
221     if x <= TwoM27 {
222         return U00 + (2/Pi)*Log(x) // |x| < ~7.4506e
223     }
224     z := x * x
225     u := U00 + z*(U01+z*(U02+z*(U03+z*(U04+z*(U05+z*U06)
226     v := 1 + z*(V01+z*(V02+z*(V03+z*V04)))
227     return u/v + (2/Pi)*J0(x)*Log(x) // ~7.4506e-9 < |x|
228 }
229
230 // The asymptotic expansions of pzero is
231 //  $1 - 9/128 s^{**2} + 11025/98304 s^{**4} - \dots$ , where  $s = 1$ 
232 // For  $x \geq 2$ , We approximate pzero by
233 //  $pzero(x) = 1 + (R/S)$ 
234 // where  $R = pR0 + pR1*s^{**2} + pR2*s^{**4} + \dots + pR5*s^{**10}$ 
235 //  $S = 1 + pS0*s^{**2} + \dots + pS4*s^{**10}$ 
236 // and
237 //  $|pzero(x)-1-R/S| \leq 2^{**(-60.26)}$ 
238
239 // for x in [inf, 8]=1/[0,0.125]
240 var p0R8 = [6]float64{
241     0.00000000000000000000e+00, // 0x0000000000000000
242     -7.031249999999900357484e-02, // 0xBFB1FFFFFFFFFD32

```

```

243         -8.08167041275349795626e+00, // 0xC02029D0B44FA779
244         -2.57063105679704847262e+02, // 0xC07011027B19E863
245         -2.48521641009428822144e+03, // 0xC0A36A6ECD4DCAFC
246         -5.25304380490729545272e+03, // 0xC0B4850B36CC643D
247     }
248     var p0S8 = [5]float64{
249         1.16534364619668181717e+02, // 0x405D223307A96751
250         3.83374475364121826715e+03, // 0x40ADF37D50596938
251         4.05978572648472545552e+04, // 0x40E3D2BB6EB6B05F
252         1.16752972564375915681e+05, // 0x40FC810F8F9FA9BD
253         4.76277284146730962675e+04, // 0x40E741774F2C49DC
254     }
255
256     // for x in [8,4.5454]=1/[0.125,0.22001]
257     var p0R5 = [6]float64{
258         -1.14125464691894502584e-11, // 0xBDA918B147E495CC
259         -7.03124940873599280078e-02, // 0xBFB1FFFFE69AFBC6
260         -4.15961064470587782438e+00, // 0xC010A370F90C6BBF
261         -6.76747652265167261021e+01, // 0xC050EB2F5A7D1783
262         -3.31231299649172967747e+02, // 0xC074B3B36742CC63
263         -3.46433388365604912451e+02, // 0xC075A6EF28A38BD7
264     }
265     var p0S5 = [5]float64{
266         6.07539382692300335975e+01, // 0x404E60810C98C5DE
267         1.05125230595704579173e+03, // 0x40906D025C7E2864
268         5.97897094333855784498e+03, // 0x40B75AF88FBE1D60
269         9.62544514357774460223e+03, // 0x40C2CCB8FA76FA38
270         2.40605815922939109441e+03, // 0x40A2CC1DC70BE864
271     }
272
273     // for x in [4.547,2.8571]=1/[0.2199,0.35001]
274     var p0R3 = [6]float64{
275         -2.54704601771951915620e-09, // 0xBE25E1036FE1AA86
276         -7.03119616381481654654e-02, // 0xBFB1FFF6F7C0E24B
277         -2.40903221549529611423e+00, // 0xC00345B2AEA48074
278         -2.19659774734883086467e+01, // 0xC035F74A4CB94E14
279         -5.80791704701737572236e+01, // 0xC04D0A22420A1A45
280         -3.14479470594888503854e+01, // 0xC03F72ACA892D80F
281     }
282     var p0S3 = [5]float64{
283         3.58560338055209726349e+01, // 0x4041ED9284077DD3
284         3.61513983050303863820e+02, // 0x40769839464A7C0E
285         1.19360783792111533330e+03, // 0x4092A66E6D1061D6
286         1.12799679856907414432e+03, // 0x40919FFCB8C39B7E
287         1.73580930813335754692e+02, // 0x4065B296FC379081
288     }
289
290     // for x in [2.8570,2]=1/[0.3499,0.5]
291     var p0R2 = [6]float64{

```

```

292         -8.87534333032526411254e-08, // 0xBE77D316E927026D
293         -7.03030995483624743247e-02, // 0xBF61FF62495E1E42
294         -1.45073846780952986357e+00, // 0xBFF736398A24A843
295         -7.63569613823527770791e+00, // 0xC01E8AF3EDAF7F3
296         -1.11931668860356747786e+01, // 0xC02662E6C5246303
297         -3.23364579351335335033e+00, // 0xC009DE81AF8FE70F
298     }
299     var p0S2 = [5]float64{
300         2.22202997532088808441e+01, // 0x40363865908B5959
301         1.36206794218215208048e+02, // 0x4061069E0EE8878F
302         2.70470278658083486789e+02, // 0x4070E78642EA079B
303         1.53875394208320329881e+02, // 0x40633C033AB6FAFF
304         1.46576176948256193810e+01, // 0x402D50B344391809
305     }
306
307     func pzero(x float64) float64 {
308         var p [6]float64
309         var q [5]float64
310         if x >= 8 {
311             p = p0R8
312             q = p0S8
313         } else if x >= 4.5454 {
314             p = p0R5
315             q = p0S5
316         } else if x >= 2.8571 {
317             p = p0R3
318             q = p0S3
319         } else if x >= 2 {
320             p = p0R2
321             q = p0S2
322         }
323         z := 1 / (x * x)
324         r := p[0] + z*(p[1]+z*(p[2]+z*(p[3]+z*(p[4]+z*p[5])))
325         s := 1 + z*(q[0]+z*(q[1]+z*(q[2]+z*(q[3]+z*q[4])))
326         return 1 + r/s
327     }
328
329     // For x >= 8, the asymptotic expansions of qzero is
330     //     -1/8 s + 75/1024 s**3 - ..., where s = 1/x.
331     // We approximate pzero by
332     //     qzero(x) = s*(-1.25 + (R/S))
333     // where R = qR0 + qR1*s**2 + qR2*s**4 + ... + qR5*s**10
334     //     S = 1 + qS0*s**2 + ... + qS5*s**12
335     // and
336     //     | qzero(x)/s +1.25-R/S | <= 2**(-61.22)
337
338     // for x in [inf, 8]=1/[0,0.125]
339     var q0R8 = [6]float64{
340         0.00000000000000000000e+00, // 0x0000000000000000

```

```

341         7.32421874999935051953e-02, // 0x3FB2BFFFFFFFFFE2C
342         1.17682064682252693899e+01, // 0x402789525BB334D6
343         5.57673380256401856059e+02, // 0x40816D6315301825
344         8.85919720756468632317e+03, // 0x40C14D993E18F46D
345         3.70146267776887834771e+04, // 0x40E212D40E901566
346     }
347     var q0S8 = [6]float64{
348         1.63776026895689824414e+02, // 0x406478D5365B39BC
349         8.09834494656449805916e+03, // 0x40BFA2584E6B0563
350         1.42538291419120476348e+05, // 0x4101665254D38C3F
351         8.03309257119514397345e+05, // 0x412883DA83A52B43
352         8.40501579819060512818e+05, // 0x4129A66B28DE0B3D
353         -3.43899293537866615225e+05, // 0xC114FD6D2C9530C5
354     }
355
356     // for x in [8,4.5454]=1/[0.125,0.22001]
357     var q0R5 = [6]float64{
358         1.84085963594515531381e-11, // 0x3DB43D8F29CC8CD9
359         7.32421766612684765896e-02, // 0x3FB2BFFFD172B04C
360         5.83563508962056953777e+00, // 0x401757B0B9953DD3
361         1.35111577286449829671e+02, // 0x4060E3920A8788E9
362         1.02724376596164097464e+03, // 0x40900CF99DC8C481
363         1.98997785864605384631e+03, // 0x409F17E953C6E3A6
364     }
365     var q0S5 = [6]float64{
366         8.27766102236537761883e+01, // 0x4054B1B3FB5E1543
367         2.07781416421392987104e+03, // 0x40A03BA0DA21C0CE
368         1.88472887785718085070e+04, // 0x40D267D27B591E6D
369         5.67511122894947329769e+04, // 0x40EBB5E397E02372
370         3.59767538425114471465e+04, // 0x40E191181F7A54A0
371         -5.35434275601944773371e+03, // 0xC0B4EA57BEDBC609
372     }
373
374     // for x in [4.547,2.8571]=1/[0.2199,0.35001]
375     var q0R3 = [6]float64{
376         4.37741014089738620906e-09, // 0x3E32CD036ADEC8B2
377         7.32411180042911447163e-02, // 0x3FB2BFEE0E8D0842
378         3.34423137516170720929e+00, // 0x400AC0FC61149CF5
379         4.26218440745412650017e+01, // 0x40454F98962DAEDD
380         1.70808091340565596283e+02, // 0x406559DBE25EFD1F
381         1.66733948696651168575e+02, // 0x4064D77C81FA21E0
382     }
383     var q0S3 = [6]float64{
384         4.87588729724587182091e+01, // 0x40486122BFE343A6
385         7.09689221056606015736e+02, // 0x40862D8386544EB3
386         3.70414822620111362994e+03, // 0x40ACF04BE44DFC63
387         6.46042516752568917582e+03, // 0x40B93C6CD7C76A28
388         2.51633368920368957333e+03, // 0x40A3A8AAD94FB1C0
389         -1.49247451836156386662e+02, // 0xC062A7EB201CF40F
390     }

```

```

391
392 // for x in [2.8570,2]=1/[0.3499,0.5]
393 var q0R2 = [6]float64{
394     1.504444444886983272379e-07, // 0x3E84313B54F76BDB
395     7.32234265963079278272e-02, // 0x3FB2BEC53E883E34
396     1.99819174093815998816e+00, // 0x3FFFF897E727779C
397     1.44956029347885735348e+01, // 0x402CFDBFAAF96FE5
398     3.16662317504781540833e+01, // 0x403FAA8E29FBDC4A
399     1.62527075710929267416e+01, // 0x403040B171814BB4
400 }
401 var q0S2 = [6]float64{
402     3.03655848355219184498e+01, // 0x403E5D96F7C07AED
403     2.69348118608049844624e+02, // 0x4070D591E4D14B40
404     8.44783757595320139444e+02, // 0x408A664522B3BF22
405     8.82935845112488550512e+02, // 0x408B977C9C5CC214
406     2.12666388511798828631e+02, // 0x406A95530E001365
407     -5.31095493882666946917e+00, // 0xC0153E6AF8B32931
408 }
409
410 func qzero(x float64) float64 {
411     var p, q [6]float64
412     if x >= 8 {
413         p = q0R8
414         q = q0S8
415     } else if x >= 4.5454 {
416         p = q0R5
417         q = q0S5
418     } else if x >= 2.8571 {
419         p = q0R3
420         q = q0S3
421     } else if x >= 2 {
422         p = q0R2
423         q = q0S2
424     }
425     z := 1 / (x * x)
426     r := p[0] + z*(p[1]+z*(p[2]+z*(p[3]+z*(p[4]+z*p[5])))
427     s := 1 + z*(q[0]+z*(q[1]+z*(q[2]+z*(q[3]+z*(q[4]+z*q
428     return (-0.125 + r/s) / x
429 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/j1.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8      Bessel function of the first and second kinds of order
9 */
10
11 // The original C code and the long comment below are
12 // from FreeBSD's /usr/src/lib/msun/src/e_j1.c and
13 // came with this notice. The go code is a simplified
14 // version of the original C.
15 //
16 // =====
17 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
18 //
19 // Developed at SunPro, a Sun Microsystems, Inc. business.
20 // Permission to use, copy, modify, and distribute this
21 // software is freely granted, provided that this notice
22 // is preserved.
23 // =====
24 //
25 // __ieee754_j1(x), __ieee754_y1(x)
26 // Bessel function of the first and second kinds of order on
27 // Method -- j1(x):
28 //     1. For tiny x, we use  $j_1(x) = x/2 - x^3/16 + x^5/3$ 
29 //     2. Reduce x to |x| since  $j_1(x) = -j_1(-x)$ , and
30 //     for x in (0,2)
31 //          $j_1(x) = x/2 + x^2 * R0/S0$ , where  $z = x^2$ ;
32 //     (precision:  $|j_1(x) - 1/2 - R0/S0| < 2^{-61.51}$ )
33 //     for x in (2,inf)
34 //          $j_1(x) = \sqrt{2/(pi*x)} * (p_1(x) * \cos(x_1) - q_1(x) * \sin(x_1))$ 
35 //          $y_1(x) = \sqrt{2/(pi*x)} * (p_1(x) * \sin(x_1) + q_1(x) * \cos(x_1))$ 
36 //     where  $x_1 = x - 3*pi/4$ . It is better to compute sin(x_1)
37 //     as follow:
38 //          $\cos(x_1) = \cos(x)\cos(3pi/4) + \sin(x)\sin(3pi/4)$ 
39 //          $= 1/\sqrt{2} * (\sin(x) - \cos(x))$ 
40 //          $\sin(x_1) = \sin(x)\cos(3pi/4) - \cos(x)\sin(3pi/4)$ 
41 //          $= -1/\sqrt{2} * (\sin(x) + \cos(x))$ 
42 //     (To avoid cancellation, use
43 //          $\sin(x) + \cos(x) = -\cos(2x)/(\sin(x) - \cos(x))$ 
44 //     to compute the worse one.)
```

```

45 //
46 //      3 Special cases
47 //          j1(nan)= nan
48 //          j1(0) = 0
49 //          j1(inf) = 0
50 //
51 // Method -- y1(x):
52 //      1. screen out x<=0 cases: y1(0)=-inf, y1(x<0)=NaN
53 //      2. For x<2.
54 //          Since
55 //               $y_1(x) = 2/\pi * (j_1(x) * (\ln(x/2) + \text{Euler}) - 1/x - x/2 +$ 
56 //          therefore  $y_1(x) - 2/\pi * j_1(x) * \ln(x) - 1/x$  is an odd fu
57 //          We use the following function to approximate y1,
58 //               $y_1(x) = x * U(z) / V(z) + (2/\pi) * (j_1(x) * \ln(x) - 1/x$ 
59 //          where for x in [0,2] (abs err less than 2**-65.89
60 //               $U(z) = U_0[0] + U_0[1]*z + \dots + U_0[4]*z^{**4}$ 
61 //               $V(z) = 1 + v_0[0]*z + \dots + v_0[4]*z^{**5}$ 
62 //          Note: For tiny x, 1/x dominate y1 and hence
63 //               $y_1(\text{tiny}) = -2/\pi / \text{tiny}$ , (choose tiny < 2**-54)
64 //      3. For x >= 2.
65 //               $y_1(x) = \sqrt{2/(\pi * x)} * (p_1(x) * \sin(x_1) + q_1(x)$ 
66 //          where  $x_1 = x - 3 * \pi / 4$ . It is better to compute sin(
67 //          by method mentioned above.
68
69 // J1 returns the order-one Bessel function of the first kin
70 //
71 // Special cases are:
72 //      J1(±Inf) = 0
73 //      J1(NaN) = NaN
74 func J1(x float64) float64 {
75     const (
76         TwoM27 = 1.0 / (1 << 27) // 2**-27 0x3e40000
77         Two129 = 1 << 129 // 2**129 0x4800000
78         // R0/S0 on [0, 2]
79         R00 = -6.2500000000000000000000e-02 // 0xBF800
80         R01 = 1.40705666955189706048e-03 // 0x3F570
81         R02 = -1.59955631084035597520e-05 // 0xBEF0C
82         R03 = 4.96727999609584448412e-08 // 0x3E6AA
83         S01 = 1.91537599538363460805e-02 // 0x3F939
84         S02 = 1.85946785588630915560e-04 // 0x3F285
85         S03 = 1.17718464042623683263e-06 // 0x3EB3E
86         S04 = 5.04636257076217042715e-09 // 0x3E35A
87         S05 = 1.23542274426137913908e-11 // 0x3DAB2
88     )
89     // special cases
90     switch {
91     case IsNaN(x):
92         return x
93     case IsInf(x, 0) || x == 0:
94         return 0

```

```

95     }
96
97     sign := false
98     if x < 0 {
99         x = -x
100        sign = true
101    }
102    if x >= 2 {
103        s, c := Sincos(x)
104        ss := -s - c
105        cc := s - c
106
107        // make sure x+x does not overflow
108        if x < MaxFloat64/2 {
109            z := Cos(x + x)
110            if s*c > 0 {
111                cc = z / ss
112            } else {
113                ss = z / cc
114            }
115        }
116
117        // j1(x) = 1/sqrt(pi) * (P(1,x)*cc - Q(1,x)*
118        // y1(x) = 1/sqrt(pi) * (P(1,x)*ss + Q(1,x)*
119
120        var z float64
121        if x > Two129 {
122            z = (1 / SqrtPi) * cc / Sqrt(x)
123        } else {
124            u := pone(x)
125            v := gone(x)
126            z = (1 / SqrtPi) * (u*cc - v*ss) / S
127        }
128        if sign {
129            return -z
130        }
131        return z
132    }
133    if x < TwoM27 { // |x|<2**-27
134        return 0.5 * x // inexact if x!=0 necessary
135    }
136    z := x * x
137    r := z * (R00 + z*(R01+z*(R02+z*R03)))
138    s := 1.0 + z*(S01+z*(S02+z*(S03+z*(S04+z*S05))))
139    r *= x
140    z = 0.5*x + r/s
141    if sign {
142        return -z
143    }

```

```

144         return z
145     }
146
147     // Y1 returns the order-one Bessel function of the second ki
148     //
149     // Special cases are:
150     //     Y1(+Inf) = 0
151     //     Y1(0) = -Inf
152     //     Y1(x < 0) = NaN
153     //     Y1(NaN) = NaN
154     func Y1(x float64) float64 {
155         const (
156             TwoM54 = 1.0 / (1 << 54)           // 2**-
157             Two129 = 1 << 129                 // 2**1
158             U00    = -1.96057090646238940668e-01 // 0xBF
159             U01    = 5.04438716639811282616e-02  // 0x3F
160             U02    = -1.91256895875763547298e-03 // 0xBF
161             U03    = 2.35252600561610495928e-05  // 0x3E
162             U04    = -9.19099158039878874504e-08 // 0xBE
163             V00    = 1.99167318236649903973e-02  // 0x3F
164             V01    = 2.02552581025135171496e-04  // 0x3F
165             V02    = 1.35608801097516229404e-06  // 0x3E
166             V03    = 6.22741452364621501295e-09  // 0x3E
167             V04    = 1.66559246207992079114e-11  // 0x3D
168         )
169         // special cases
170         switch {
171         case x < 0 || IsNaN(x):
172             return NaN()
173         case IsInf(x, 1):
174             return 0
175         case x == 0:
176             return Inf(-1)
177         }
178
179         if x >= 2 {
180             s, c := Sincos(x)
181             ss := -s - c
182             cc := s - c
183
184             // make sure x+x does not overflow
185             if x < MaxFloat64/2 {
186                 z := Cos(x + x)
187                 if s*c > 0 {
188                     cc = z / ss
189                 } else {
190                     ss = z / cc
191                 }
192             }

```

```

193 // y1(x) = sqrt(2/(pi*x))*(p1(x)*sin(x0)+q1(
194 // where x0 = x-3pi/4
195 // Better formula:
196 // cos(x0) = cos(x)cos(3pi/4)+sin(x)
197 // = 1/sqrt(2) * (sin(x) -
198 // sin(x0) = sin(x)cos(3pi/4)-cos(x)
199 // = -1/sqrt(2) * (cos(x) +
200 // To avoid cancellation, use
201 // sin(x) +- cos(x) = -cos(2x)/(sin(x) -
202 // to compute the worse one.
203
204 var z float64
205 if x > Two129 {
206     z = (1 / SqrtPi) * ss / Sqrt(x)
207 } else {
208     u := pone(x)
209     v := qone(x)
210     z = (1 / SqrtPi) * (u*ss + v*cc) / S
211 }
212 return z
213 }
214 if x <= TwoM54 { // x < 2**-54
215     return -(2 / Pi) / x
216 }
217 z := x * x
218 u := U00 + z*(U01+z*(U02+z*(U03+z*U04)))
219 v := 1 + z*(V00+z*(V01+z*(V02+z*(V03+z*V04))))
220 return x*(u/v) + (2/Pi)*(J1(x)*Log(x)-1/x)
221 }
222
223 // For x >= 8, the asymptotic expansions of pone is
224 // 1 + 15/128 s**2 - 4725/2**15 s**4 - ..., where s = 1
225 // We approximate pone by
226 // pone(x) = 1 + (R/S)
227 // where R = pr0 + pr1*s**2 + pr2*s**4 + ... + pr5*s**10
228 // S = 1 + ps0*s**2 + ... + ps4*s**10
229 // and
230 // | pone(x)-1-R/S | <= 2**(-60.06)
231
232 // for x in [inf, 8]=1/[0,0.125]
233 var p1R8 = [6]float64{
234     0.00000000000000000000e+00, // 0x00000000000000000000
235     1.17187499999988647970e-01, // 0x3FBDFEEEEEEFFCCE
236     1.32394806593073575129e+01, // 0x402A7A9D357F7FCE
237     4.12051854307378562225e+02, // 0x4079C0D4652EA590
238     3.87474538913960532227e+03, // 0x40AE457DA3A532CC
239     7.91447954031891731574e+03, // 0x40BEEA7AC32782DD
240 }
241 var p1S8 = [5]float64{
242     1.14207370375678408436e+02, // 0x405C8D458E656CAC

```

```

243         3.65093083420853463394e+03, // 0x40AC85DC964D274F
244         3.69562060269033463555e+04, // 0x40E20B8697C5BB7F
245         9.76027935934950801311e+04, // 0x40F7D42CB28F17BB
246         3.08042720627888811578e+04, // 0x40DE1511697A0B2D
247     }
248
249     // for x in [8,4.5454] = 1/[0.125,0.22001]
250     var p1R5 = [6]float64{
251         1.31990519556243522749e-11, // 0x3DAD0667DAE1CA7D
252         1.17187493190614097638e-01, // 0x3FBDFFFFE2C10043
253         6.80275127868432871736e+00, // 0x401B36046E6315E3
254         1.08308182990189109773e+02, // 0x405B13B9452602ED
255         5.17636139533199752805e+02, // 0x40802D16D052D649
256         5.28715201363337541807e+02, // 0x408085B8BB7E0CB7
257     }
258     var p1S5 = [5]float64{
259         5.92805987221131331921e+01, // 0x404DA3EAA8AF633D
260         9.91401418733614377743e+02, // 0x408EFB361B066701
261         5.35326695291487976647e+03, // 0x40B4E9445706B6FB
262         7.84469031749551231769e+03, // 0x40BEA4B0B8A5BB15
263         1.50404688810361062679e+03, // 0x40978030036F5E51
264     }
265
266     // for x in [4.5453,2.8571] = 1/[0.2199,0.35001]
267     var p1R3 = [6]float64{
268         3.02503916137373618024e-09, // 0x3E29FC21A7AD9EDD
269         1.17186865567253592491e-01, // 0x3FBDFFF55B21D17B
270         3.93297750033315640650e+00, // 0x400F76BCE85EAD8A
271         3.51194035591636932736e+01, // 0x40418F489DA6D129
272         9.10550110750781271918e+01, // 0x4056C3854D2C1837
273         4.85590685197364919645e+01, // 0x4048478F8EA83EE5
274     }
275     var p1S3 = [5]float64{
276         3.47913095001251519989e+01, // 0x40416549A134069C
277         3.36762458747825746741e+02, // 0x40750C3307F1A75F
278         1.04687139975775130551e+03, // 0x40905B7C5037D523
279         8.90811346398256432622e+02, // 0x408BD67DA32E31E9
280         1.03787932439639277504e+02, // 0x4059F26D7C2EED53
281     }
282
283     // for x in [2.8570,2] = 1/[0.3499,0.5]
284     var p1R2 = [6]float64{
285         1.07710830106873743082e-07, // 0x3E7CE9D4F65544F4
286         1.17176219462683348094e-01, // 0x3FBDFFF42BE760D83
287         2.36851496667608785174e+00, // 0x4002F2B7F98FAEC0
288         1.22426109148261232917e+01, // 0x40287C377F71A964
289         1.76939711271687727390e+01, // 0x4031B1A8177F8EE2
290         5.07352312588818499250e+00, // 0x40144B49A574C1FE
291     }

```

```

292 var p1S2 = [5]float64{
293     2.14364859363821409488e+01, // 0x40356FBD8AD5ECDC
294     1.25290227168402751090e+02, // 0x405F529314F92CD5
295     2.32276469057162813669e+02, // 0x406D08D8D5A2DBD9
296     1.17679373287147100768e+02, // 0x405D6B7ADA1884A9
297     8.36463893371618283368e+00, // 0x4020BAB1F44E5192
298 }
299
300 func pone(x float64) float64 {
301     var p [6]float64
302     var q [5]float64
303     if x >= 8 {
304         p = p1R8
305         q = p1S8
306     } else if x >= 4.5454 {
307         p = p1R5
308         q = p1S5
309     } else if x >= 2.8571 {
310         p = p1R3
311         q = p1S3
312     } else if x >= 2 {
313         p = p1R2
314         q = p1S2
315     }
316     z := 1 / (x * x)
317     r := p[0] + z*(p[1]+z*(p[2]+z*(p[3]+z*(p[4]+z*p[5])))
318     s := 1.0 + z*(q[0]+z*(q[1]+z*(q[2]+z*(q[3]+z*q[4])))
319     return 1 + r/s
320 }
321
322 // For x >= 8, the asymptotic expansions of qone is
323 //     3/8 s - 105/1024 s**3 - ..., where s = 1/x.
324 // We approximate qone by
325 //     qone(x) = s*(0.375 + (R/S))
326 // where R = qr1*s**2 + qr2*s**4 + ... + qr5*s**10
327 //     S = 1 + qs1*s**2 + ... + qs6*s**12
328 // and
329 //     | qone(x)/s - 0.375 - R/S | <= 2**(-61.13)
330
331 // for x in [inf, 8] = 1/[0,0.125]
332 var q1R8 = [6]float64{
333     0.00000000000000000000e+00, // 0x00000000000000000000
334     -1.02539062499992714161e-01, // 0xBFBA3FFFFFFFFDF3
335     -1.62717534544589987888e+01, // 0xC0304591A26779F7
336     -7.59601722513950107896e+02, // 0xC087BCD053E4B576
337     -1.18498066702429587167e+04, // 0xC0C724E740F87415
338     -4.84385124285750353010e+04, // 0xC0E7A6D065D09C6A
339 }
340 var q1S8 = [6]float64{

```

```

341         1.61395369700722909556e+02, // 0x40642CA6DE5BCDE5
342         7.82538599923348465381e+03, // 0x40BE9162D0D88419
343         1.33875336287249578163e+05, // 0x4100579AB0B75E98
344         7.19657723683240939863e+05, // 0x4125F65372869C19
345         6.66601232617776375264e+05, // 0x412457D27719AD5C
346         -2.94490264303834643215e+05, // 0xC111F9690EA5AA18
347     }
348
349     // for x in [8,4.5454] = 1/[0.125,0.22001]
350     var q1R5 = [6]float64{
351         -2.08979931141764104297e-11, // 0xBDB6FA431AA1A098
352         -1.02539050241375426231e-01, // 0xBFBA3FFFCB597FEF
353         -8.05644828123936029840e+00, // 0xC0201CE6CA03AD4B
354         -1.83669607474888380239e+02, // 0xC066F56D6CA7B9B0
355         -1.37319376065508163265e+03, // 0xC09574C66931734F
356         -2.61244440453215656817e+03, // 0xC0A468E388FDA79D
357     }
358     var q1S5 = [6]float64{
359         8.12765501384335777857e+01, // 0x405451B2FF5A11B2
360         1.99179873460485964642e+03, // 0x409F1F31E77BF839
361         1.74684851924908907677e+04, // 0x40D10F1F0D64CE29
362         4.98514270910352279316e+04, // 0x40E8576DAABAD197
363         2.79480751638918118260e+04, // 0x40DB4B04CF7C364B
364         -4.71918354795128470869e+03, // 0xC0B26F2EFCFFA004
365     }
366
367     // for x in [4.5454,2.8571] = 1/[0.2199,0.35001] ???
368     var q1R3 = [6]float64{
369         -5.07831226461766561369e-09, // 0xBE35CFA9D38FC84F
370         -1.02537829820837089745e-01, // 0xBFBA3FEB51AEED54
371         -4.61011581139473403113e+00, // 0xC01270C23302D9FF
372         -5.78472216562783643212e+01, // 0xC04CEC71C25D16DA
373         -2.28244540737631695038e+02, // 0xC06C87D34718D55F
374         -2.19210128478909325622e+02, // 0xC06B66B95F5C1BF6
375     }
376     var q1S3 = [6]float64{
377         4.76651550323729509273e+01, // 0x4047D523CCD367E4
378         6.73865112676699709482e+02, // 0x40850EEBC031EE3E
379         3.38015286679526343505e+03, // 0x40AA684E448E7C9A
380         5.54772909720722782367e+03, // 0x40B5ABBAA61D54A6
381         1.90311919338810798763e+03, // 0x409DBC7A0DD4DF4B
382         -1.35201191444307340817e+02, // 0xC060E670290A311F
383     }
384
385     // for x in [2.8570,2] = 1/[0.3499,0.5]
386     var q1R2 = [6]float64{
387         -1.78381727510958865572e-07, // 0xBE87F12644C626D2
388         -1.02517042607985553460e-01, // 0xBFBA3E8E9148B010
389         -2.75220568278187460720e+00, // 0xC006048469BB4EDA
390         -1.96636162643703720221e+01, // 0xC033A9E2C168907F

```

```

391         -4.23253133372830490089e+01, // 0xC04529A3DE104AAA
392         -2.13719211703704061733e+01, // 0xC0355F3639CF6E52
393     }
394     var q1S2 = [6]float64{
395         2.95333629060523854548e+01, // 0x403D888A78AE64FF
396         2.52981549982190529136e+02, // 0x406F9F68DB821CBA
397         7.57502834868645436472e+02, // 0x4087AC05CE49A0F7
398         7.39393205320467245656e+02, // 0x40871B2548D4C029
399         1.55949003336666123687e+02, // 0x40637E5E3C3ED8D4
400         -4.95949898822628210127e+00, // 0xC013D686E71BE86B
401     }
402
403     func qone(x float64) float64 {
404         var p, q [6]float64
405         if x >= 8 {
406             p = q1R8
407             q = q1S8
408         } else if x >= 4.5454 {
409             p = q1R5
410             q = q1S5
411         } else if x >= 2.8571 {
412             p = q1R3
413             q = q1S3
414         } else if x >= 2 {
415             p = q1R2
416             q = q1S2
417         }
418         z := 1 / (x * x)
419         r := p[0] + z*(p[1]+z*(p[2]+z*(p[3]+z*(p[4]+z*p[5])))
420         s := 1 + z*(q[0]+z*(q[1]+z*(q[2]+z*(q[3]+z*(q[4]+z*q
421         return (0.375 + r/s) / x
422     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/jn.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8      Bessel function of the first and second kinds of order n
9 */
10
11 // The original C code and the long comment below are
12 // from FreeBSD's /usr/src/lib/msun/src/e_jn.c and
13 // came with this notice. The go code is a simplified
14 // version of the original C.
15 //
16 // =====
17 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
18 //
19 // Developed at SunPro, a Sun Microsystems, Inc. business.
20 // Permission to use, copy, modify, and distribute this
21 // software is freely granted, provided that this notice
22 // is preserved.
23 // =====
24 //
25 // __ieee754_jn(n, x), __ieee754_yn(n, x)
26 // floating point Bessel's function of the 1st and 2nd kind
27 // of order n
28 //
29 // Special cases:
30 //      y0(0)=y1(0)=yn(n,0) = -inf with division by zero signal
31 //      y0(-ve)=y1(-ve)=yn(n,-ve) are NaN with invalid signal
32 // Note 2. About jn(n,x), yn(n,x)
33 //      For n=0, j0(x) is called,
34 //      for n=1, j1(x) is called,
35 //      for n<x, forward recursion is used starting
36 //      from values of j0(x) and j1(x).
37 //      for n>x, a continued fraction approximation to
38 //      j(n,x)/j(n-1,x) is evaluated and then backward
39 //      recursion is used starting from a supposed value
40 //      for j(n,x). The resulting value of j(0,x) is
41 //      compared with the actual value to correct the
42 //      supposed value of j(n,x).
43 //
44 //      yn(n,x) is similar in all respects, except
```

```

45 //      that forward recursion is used for all
46 //      values of n>1.
47
48 // Jn returns the order-n Bessel function of the first kind.
49 //
50 // Special cases are:
51 //      Jn(n, ±Inf) = 0
52 //      Jn(n, NaN) = NaN
53 func Jn(n int, x float64) float64 {
54     const (
55         TwoM29 = 1.0 / (1 << 29) // 2** -29 0x3e10000
56         Two302 = 1 << 302       // 2** 302 0x52D0000
57     )
58     // special cases
59     switch {
60     case IsNaN(x):
61         return x
62     case IsInf(x, 0):
63         return 0
64     }
65     // J(-n, x) = (-1)**n * J(n, x), J(n, -x) = (-1)**n
66     // Thus, J(-n, x) = J(n, -x)
67
68     if n == 0 {
69         return J0(x)
70     }
71     if x == 0 {
72         return 0
73     }
74     if n < 0 {
75         n, x = -n, -x
76     }
77     if n == 1 {
78         return J1(x)
79     }
80     sign := false
81     if x < 0 {
82         x = -x
83         if n&1 == 1 {
84             sign = true // odd n and negative x
85         }
86     }
87     var b float64
88     if float64(n) <= x {
89         // Safe to use J(n+1,x)=2n/x *J(n,x)-J(n-1,x)
90         if x >= Two302 { // x > 2**302
91
92             // (x >> n**2)
93             //      Jn(x) = cos(x-(2n+1)*pi/
94             //      Yn(x) = sin(x-(2n+1)*pi/

```

```

95          //          Let s=sin(x), c=cos(x),
96          //          xn=x-(2n+1)*pi/4, sq
97          //
98          //          n      sin(xn)*sqrt2
99          //          -----
100         //          0      s-c
101         //          1      -s-c
102         //          2      -s+c
103         //          3      s+c
104
105         var temp float64
106         switch n & 3 {
107         case 0:
108             temp = Cos(x) + Sin(x)
109         case 1:
110             temp = -Cos(x) + Sin(x)
111         case 2:
112             temp = -Cos(x) - Sin(x)
113         case 3:
114             temp = Cos(x) - Sin(x)
115         }
116         b = (1 / SqrtPi) * temp / Sqrt(x)
117     } else {
118         b = J1(x)
119         for i, a := 1, J0(x); i < n; i++ {
120             a, b = b, b*(float64(i+i)/x)
121         }
122     }
123 } else {
124     if x < TwoM29 { // x < 2** -29
125         // x is tiny, return the first Taylor
126         // J(n,x) = 1/n!*(x/2)**n - ...
127
128         if n > 33 { // underflow
129             b = 0
130         } else {
131             temp := x * 0.5
132             b = temp
133             a := 1.0
134             for i := 2; i <= n; i++ {
135                 a *= float64(i) // a
136                 b *= temp      // b
137             }
138             b /= a
139         }
140     } else {
141         // use backward recurrence
142         //          x      x**2
143         // J(n,x)/J(n-1,x) = ----- -----

```

```

144 //          2n - 2(n+1)
145 //
146 //          1      1
147 // (for large x) = ---- ----
148 //          2n    2(n+1)
149 //          - - - - -
150 //          x      x
151 //
152 // Let w = 2n/x and h=2/x, then the
153 // is equal to the continued fractio
154 //          1
155 //          = -----
156 //          1
157 //          w - -----
158 //          1
159 //          w+h - -----
160 //          w+2h - ...
161 //
162 // To determine how many terms neede
163 // Q(0) = w, Q(1) = w(w+h) - 1,
164 // Q(k) = (w+k*h)*Q(k-1) - Q(k-2),
165 // When Q(k) > 1e4      good for sin
166 // When Q(k) > 1e9      good for dou
167 // When Q(k) > 1e17     good for qua
168
169 // determine k
170 w := float64(n+n) / x
171 h := 2 / x
172 q0 := w
173 z := w + h
174 q1 := w*z - 1
175 k := 1
176 for q1 < 1e9 {
177     k += 1
178     z += h
179     q0, q1 = q1, z*q1-q0
180 }
181 m := n + n
182 t := 0.0
183 for i := 2 * (n + k); i >= m; i -= 2
184     t = 1 / (float64(i)/x - t)
185 }
186 a := t
187 b = 1
188 // estimate log((2/x)**n*n!) = n*lo
189 // Hence, if n*(log(2n/x)) > ...
190 // single 8.8722839355e+01
191 // double 7.09782712893383973096e+0
192 // long double 1.135652340629414394

```

```

193 // then recurrent value may overflo
194 // likely underflow to zero
195
196 tmp := float64(n)
197 v := 2 / x
198 tmp = tmp * Log(Abs(v*tmp))
199 if tmp < 7.09782712893383973096e+02
200     for i := n - 1; i > 0; i-- {
201         di := float64(i + i)
202         a, b = b, b*di/x-a
203         di -= 2
204     }
205 } else {
206     for i := n - 1; i > 0; i-- {
207         di := float64(i + i)
208         a, b = b, b*di/x-a
209         di -= 2
210         // scale b to avoid
211         if b > 1e100 {
212             a /= b
213             t /= b
214             b = 1
215         }
216     }
217 }
218 b = t * J0(x) / b
219 }
220 }
221 if sign {
222     return -b
223 }
224 return b
225 }
226
227 // Yn returns the order-n Bessel function of the second kind
228 //
229 // Special cases are:
230 //     Yn(n, +Inf) = 0
231 //     Yn(n > 0, 0) = -Inf
232 //     Yn(n < 0, 0) = +Inf if n is odd, -Inf if n is even
233 //     Y1(n, x < 0) = NaN
234 //     Y1(n, NaN) = NaN
235 func Yn(n int, x float64) float64 {
236     const Two302 = 1 << 302 // 2**302 0x52D0000000000000
237     // special cases
238     switch {
239     case x < 0 || IsNaN(x):
240         return NaN()
241     case IsInf(x, 1):
242         return 0

```

```

243     }
244
245     if n == 0 {
246         return Y0(x)
247     }
248     if x == 0 {
249         if n < 0 && n&1 == 1 {
250             return Inf(1)
251         }
252         return Inf(-1)
253     }
254     sign := false
255     if n < 0 {
256         n = -n
257         if n&1 == 1 {
258             sign = true // sign true if n < 0 &&
259         }
260     }
261     if n == 1 {
262         if sign {
263             return -Y1(x)
264         }
265         return Y1(x)
266     }
267     var b float64
268     if x >= Two302 { // x > 2**302
269         // (x >> n**2)
270         //         Jn(x) = cos(x-(2n+1)*pi/4)*sqrt(
271         //         Yn(x) = sin(x-(2n+1)*pi/4)*sqrt(
272         //         Let s=sin(x), c=cos(x),
273         //         xn=x-(2n+1)*pi/4, sqrt2 = sqr
274         //
275         //             n      sin(xn)*sqrt2      cos(
276         //             -----
277         //             0      s-c                  c+s
278         //             1      -s-c                 -c+s
279         //             2      -s+c                 -c-s
280         //             3      s+c                  c-s
281
282         var temp float64
283         switch n & 3 {
284         case 0:
285             temp = Sin(x) - Cos(x)
286         case 1:
287             temp = -Sin(x) - Cos(x)
288         case 2:
289             temp = -Sin(x) + Cos(x)
290         case 3:
291             temp = Sin(x) + Cos(x)

```

```
292     }
293     b = (1 / SqrtPi) * temp / Sqrt(x)
294 } else {
295     a := Y0(x)
296     b = Y1(x)
297     // quit if b is -inf
298     for i := 1; i < n && !IsInf(b, -1); i++ {
299         a, b = b, (float64(i+i)/x)*b-a
300     }
301 }
302 if sign {
303     return -b
304 }
305 return b
306 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/ldexp.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Ldexp is the inverse of Frexp.
8 // It returns frac × 2**exp.
9 //
10 // Special cases are:
11 //     Ldexp(±0, exp) = ±0
12 //     Ldexp(±Inf, exp) = ±Inf
13 //     Ldexp(NaN, exp) = NaN
14 func Ldexp(frac float64, exp int) float64
15
16 func ldexp(frac float64, exp int) float64 {
17     // special cases
18     switch {
19     case frac == 0:
20         return frac // correctly return -0
21     case IsInf(frac, 0) || IsNaN(frac):
22         return frac
23     }
24     frac, e := normalize(frac)
25     exp += e
26     x := Float64bits(frac)
27     exp += int(x>>shift)&mask - bias
28     if exp < -1074 {
29         return Copysign(0, frac) // underflow
30     }
31     if exp > 1023 { // overflow
32         if frac < 0 {
33             return Inf(-1)
34         }
35         return Inf(1)
36     }
37     var m float64 = 1
38     if exp < -1022 { // denormal
39         exp += 52
40         m = 1.0 / (1 << 52) // 2**-52
41     }
42     x ^= mask << shift
43     x |= uint64(exp+bias) << shift
44     return m * Float64frombits(x)
```

45 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/lgamma.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Floating-point logarithm of the Gamma function.
9 */
10
11 // The original C code and the long comment below are
12 // from FreeBSD's /usr/src/lib/msun/src/e_lgamma_r.c and
13 // came with this notice. The go code is a simplified
14 // version of the original C.
15 //
16 // =====
17 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
18 //
19 // Developed at SunPro, a Sun Microsystems, Inc. business.
20 // Permission to use, copy, modify, and distribute this
21 // software is freely granted, provided that this notice
22 // is preserved.
23 // =====
24 //
25 // __ieee754_lgamma_r(x, signgamp)
26 // Reentrant version of the logarithm of the Gamma function
27 // with user provided pointer for the sign of Gamma(x).
28 //
29 // Method:
30 // 1. Argument Reduction for  $0 < x \leq 8$ 
31 //    Since  $\gamma(1+s) = s \cdot \gamma(s)$ , for  $x$  in  $[0, 8]$ , we may
32 //    reduce  $x$  to a number in  $[1.5, 2.5]$  by
33 //     $\lgamma(1+s) = \log(s) + \lgamma(s)$ 
34 //    for example,
35 //     $\lgamma(7.3) = \log(6.3) + \lgamma(6.3)$ 
36 //     $= \log(6.3 \cdot 5.3) + \lgamma(5.3)$ 
37 //     $= \log(6.3 \cdot 5.3 \cdot 4.3 \cdot 3.3 \cdot 2.3) + \lgamma(2.3)$ 
38 // 2. Polynomial approximation of  $\lgamma$  around its
39 //    minimum ( $y_{\min} = 1.461632144968362245$ ) to maintain monotonicity
40 //    On  $[y_{\min} - 0.23, y_{\min} + 0.27]$  (i.e.,  $[1.23164, 1.73163]$ ),
41 //    Let  $z = x - y_{\min}$ ;
42 //     $\lgamma(x) = -1.214862905358496078218 + z^{**2} \cdot$ 
43 //     $\text{poly}(z)$  is a 14 degree polynomial.
44 // 2. Rational approximation in the primary interval  $[2, 3]$ 
```

```

45 // We use the following approximation:
46 //     s = x-2.0;
47 //     lgamma(x) = 0.5*s + s*P(s)/Q(s)
48 // with accuracy
49 //     |P/Q - (lgamma(x)-0.5s)| < 2**-61.71
50 // Our algorithms are based on the following observatio
51 //
52 //     zeta(2)-1    2    zeta(3)-1
53 // lgamma(2+s) = s*(1-Euler) + ----- * s - ----- *
54 //                                2                                3
55 //
56 //     where Euler = 0.5772156649... is the Euler constant,
57 //     is very close to 0.5.
58 //
59 // 3. For x>=8, we have
60 //     lgamma(x)~(x-0.5)log(x)-x+0.5*log(2pi)+1/(12x)-1/(36
61 // (better formula:
62 //     lgamma(x)~(x-0.5)*(log(x)-1)-.5*(log(2pi)-1) + ..
63 // Let z = 1/x, then we approximation
64 //     f(z) = lgamma(x) - (x-0.5)(log(x)-1)
65 // by
66 //     w = w0 + w1*z + w2*z3 + w3*z5 + ... + w6*z11
67 // where
68 //     |w - f(z)| < 2**-58.74
69 //
70 //
71 // 4. For negative x, since (G is gamma function)
72 //     -x*G(-x)*G(x) = pi/sin(pi*x),
73 // we have
74 //     G(x) = pi/(sin(pi*x)*(-x)*G(-x))
75 // since G(-x) is positive, sign(G(x)) = sign(sin(pi*x))
76 // Hence, for x<0, signgam = sign(sin(pi*x)) and
77 //     lgamma(x) = log(|Gamma(x)|)
78 //               = log(pi/(|x*sin(pi*x)|)) - lgamma
79 // Note: one should avoid computing pi*(-x) directly in
80 // computation of sin(pi*(-x)).
81 //
82 // 5. Special Cases
83 //     lgamma(2+s) ~ s*(1-Euler) for tiny s
84 //     lgamma(1)=lgamma(2)=0
85 //     lgamma(x) ~ -log(x) for tiny x
86 //     lgamma(0) = lgamma(inf) = inf
87 //     lgamma(-integer) = +-inf
88 //
89 //
90 //
91 var _lgamA = [...]float64{
92     7.72156649015328655494e-02, // 0x3FB3C467E37DB0C8
93     3.22467033424113591611e-01, // 0x3FD4A34CC4A60FAD
94     6.73523010531292681824e-02, // 0x3FB13E001A5562A7

```

```

95         2.05808084325167332806e-02, // 0x3F951322AC92547B
96         7.38555086081402883957e-03, // 0x3F7E404FB68FEFE8
97         2.89051383673415629091e-03, // 0x3F67ADD8CCB7926B
98         1.19270763183362067845e-03, // 0x3F538A94116F3F5D
99         5.10069792153511336608e-04, // 0x3F40B6C689B99C00
100        2.20862790713908385557e-04, // 0x3F2CF2ECED10E54D
101        1.08011567247583939954e-04, // 0x3F1C5088987DFB07
102        2.52144565451257326939e-05, // 0x3EFA7074428CFA52
103        4.48640949618915160150e-05, // 0x3F07858E90A45837
104    }
105    var _lgamR = [...]float64{
106        1.0, // placeholder
107        1.39200533467621045958e+00, // 0x3FF645A762C4AB74
108        7.21935547567138069525e-01, // 0x3FE71A1893D3DCDC
109        1.71933865632803078993e-01, // 0x3FC601EDCCFBDF27
110        1.86459191715652901344e-02, // 0x3F9317EA742ED475
111        7.77942496381893596434e-04, // 0x3F497DDACA41A95B
112        7.32668430744625636189e-06, // 0x3EDEBAF7A5B38140
113    }
114    var _lgamS = [...]float64{
115        -7.72156649015328655494e-02, // 0xBF3C467E37DB0C8
116        2.14982415960608852501e-01, // 0x3FCB848B36E20878
117        3.25778796408930981787e-01, // 0x3FD4D98F4F139F59
118        1.46350472652464452805e-01, // 0x3FC2BB9CBEE5F2F7
119        2.66422703033638609560e-02, // 0x3F9B481C7E939961
120        1.84028451407337715652e-03, // 0x3F5E26B67368F239
121        3.19475326584100867617e-05, // 0x3F00BFECDD17E945
122    }
123    var _lgamT = [...]float64{
124        4.83836122723810047042e-01, // 0x3FDEF72BC8EE38A2
125        -1.47587722994593911752e-01, // 0xBFC2E4278DC6C509
126        6.46249402391333854778e-02, // 0x3FB08B4294D5419B
127        -3.27885410759859649565e-02, // 0xBFA0C9A8DF35B713
128        1.79706750811820387126e-02, // 0x3F9266E7970AF9EC
129        -1.03142241298341437450e-02, // 0xBF851F9FBA91EC6A
130        6.10053870246291332635e-03, // 0x3F78FCE0E370E344
131        -3.68452016781138256760e-03, // 0xBF6E2EFFB3E914D7
132        2.25964780900612472250e-03, // 0x3F6282D32E15C915
133        -1.40346469989232843813e-03, // 0xBF56FE8EBF2D1AF1
134        8.81081882437654011382e-04, // 0x3F4CDF0CEF61A8E9
135        -5.38595305356740546715e-04, // 0xBF41A6109C73E0EC
136        3.15632070903625950361e-04, // 0x3F34AF6D6C0EBBF7
137        -3.12754168375120860518e-04, // 0xBF347F24ECC38C38
138        3.35529192635519073543e-04, // 0x3F35FD3EE8C2D3F4
139    }
140    var _lgamU = [...]float64{
141        -7.72156649015328655494e-02, // 0xBF3C467E37DB0C8
142        6.32827064025093366517e-01, // 0x3FE4401E8B005DFF
143        1.45492250137234768737e+00, // 0x3FF7475CD119BD6F

```

```

144         9.77717527963372745603e-01, // 0x3FEF497644EA8450
145         2.28963728064692451092e-01, // 0x3FCD4EAEF6010924
146         1.33810918536787660377e-02, // 0x3F8B678BBF2BAB09
147     }
148     var _lgamV = [...]float64{
149         1.0,
150         2.45597793713041134822e+00, // 0x4003A5D7C2BD619C
151         2.12848976379893395361e+00, // 0x40010725A42B18F5
152         7.69285150456672783825e-01, // 0x3FE89DFBE45050AF
153         1.04222645593369134254e-01, // 0x3FBAAE55D6537C88
154         3.21709242282423911810e-03, // 0x3F6A5ABB57D0CF61
155     }
156     var _lgamW = [...]float64{
157         4.18938533204672725052e-01, // 0x3FDACFE390C97D69
158         8.33333333333329678849e-02, // 0x3FB55555555553B
159         -2.7777777772877536470e-03, // 0xBF66C16C16B02E5C
160         7.93650558643019558500e-04, // 0x3F4A019F98CF38B6
161         -5.95187557450339963135e-04, // 0xBF4380CB8C0FE741
162         8.36339918996282139126e-04, // 0x3F4B67BA4CDAD5D1
163         -1.63092934096575273989e-03, // 0xBF5AB89D0B9E43E4
164     }
165
166     // Lgamma returns the natural logarithm and sign (-1 or +1)
167     //
168     // Special cases are:
169     //     Lgamma(+Inf) = +Inf
170     //     Lgamma(0) = +Inf
171     //     Lgamma(-integer) = +Inf
172     //     Lgamma(-Inf) = -Inf
173     //     Lgamma(NaN) = NaN
174     func Lgamma(x float64) (lgamma float64, sign int) {
175         const (
176             Ymin = 1.461632144968362245
177             Two52 = 1 << 52 // 0x433
178             Two53 = 1 << 53 // 0x434
179             Two58 = 1 << 58 // 0x439
180             Tiny = 1.0 / (1 << 70) // 0x3b9
181             Tc = 1.46163214496836224576e+00 // 0x3FF
182             Tf = -1.21486290535849611461e-01 // 0xBFB
183             // Tt = -(tail of Tf)
184             Tt = -3.63867699703950536541e-18 // 0xBC50C7
185         )
186         // special cases
187         sign = 1
188         switch {
189         case IsNaN(x):
190             lgamma = x
191             return
192         case IsInf(x, 0):

```

```

193         lgamma = x
194         return
195     case x == 0:
196         lgamma = Inf(1)
197         return
198     }
199
200     neg := false
201     if x < 0 {
202         x = -x
203         neg = true
204     }
205
206     if x < Tiny { // if |x| < 2** -70, return -log(|x|)
207         if neg {
208             sign = -1
209         }
210         lgamma = -Log(x)
211         return
212     }
213     var nadj float64
214     if neg {
215         if x >= Two52 { // |x| >= 2**52, must be -in
216             lgamma = Inf(1)
217             return
218         }
219         t := sinPi(x)
220         if t == 0 {
221             lgamma = Inf(1) // -integer
222             return
223         }
224         nadj = Log(Pi / Abs(t*x))
225         if t < 0 {
226             sign = -1
227         }
228     }
229
230     switch {
231     case x == 1 || x == 2: // purge off 1 and 2
232         lgamma = 0
233         return
234     case x < 2: // use lgamma(x) = lgamma(x+1) - log(x)
235         var y float64
236         var i int
237         if x <= 0.9 {
238             lgamma = -Log(x)
239             switch {
240             case x >= (Ymin - 1 + 0.27): // 0.73
241                 y = 1 - x
242                 i = 0

```

```

243         case x >= (Ymin - 1 - 0.27): // 0.23
244             y = x - (Tc - 1)
245             i = 1
246         default: // 0 < x < 0.2316
247             y = x
248             i = 2
249     }
250 } else {
251     lgamma = 0
252     switch {
253     case x >= (Ymin + 0.27): // 1.7316 <
254         y = 2 - x
255         i = 0
256     case x >= (Ymin - 0.27): // 1.2316 <
257         y = x - Tc
258         i = 1
259     default: // 0.9 < x < 1.2316
260         y = x - 1
261         i = 2
262     }
263 }
264 switch i {
265 case 0:
266     z := y * y
267     p1 := _lgamA[0] + z*(_lgamA[2]+z*(1
268     p2 := z * (_lgamA[1] + z*(+_lgamA[3]
269     p := y*p1 + p2
270     lgamma += (p - 0.5*y)
271 case 1:
272     z := y * y
273     w := z * y
274     p1 := _lgamT[0] + w*(_lgamT[3]+w*(1
275     p2 := _lgamT[1] + w*(_lgamT[4]+w*(1
276     p3 := _lgamT[2] + w*(_lgamT[5]+w*(1
277     p := z*p1 - (Tt - w*(p2+y*p3))
278     lgamma += (Tf + p)
279 case 2:
280     p1 := y * (_lgamU[0] + y*(_lgamU[1]+
281     p2 := 1 + y*(_lgamV[1]+y*(_lgamV[2]+
282     lgamma += (-0.5*y + p1/p2)
283 }
284 case x < 8: // 2 <= x < 8
285     i := int(x)
286     y := x - float64(i)
287     p := y * (_lgamS[0] + y*(_lgamS[1]+y*(_lgamS
288     q := 1 + y*(_lgamR[1]+y*(_lgamR[2]+y*(_lgamR
289     lgamma = 0.5*y + p/q
290     z := 1.0 // Lgamma(1+s) = Log(s) + Lgamma(s)
291     switch i {

```

```

292         case 7:
293             z *= (y + 6)
294             fallthrough
295         case 6:
296             z *= (y + 5)
297             fallthrough
298         case 5:
299             z *= (y + 4)
300             fallthrough
301         case 4:
302             z *= (y + 3)
303             fallthrough
304         case 3:
305             z *= (y + 2)
306             lgamma += Log(z)
307     }
308     case x < Two58: // 8 <= x < 2**58
309         t := Log(x)
310         z := 1 / x
311         y := z * z
312         w := _lgamW[0] + z*(_lgamW[1]+y*(_lgamW[2]+y
313         lgamma = (x-0.5)*(t-1) + w
314     default: // 2**58 <= x <= Inf
315         lgamma = x * (Log(x) - 1)
316     }
317     if neg {
318         lgamma = nadj - lgamma
319     }
320     return
321 }
322
323 // sinPi(x) is a helper function for negative x
324 func sinPi(x float64) float64 {
325     const (
326         Two52 = 1 << 52 // 0x4330000000000000 ~4.503
327         Two53 = 1 << 53 // 0x4340000000000000 ~9.007
328     )
329     if x < 0.25 {
330         return -Sin(Pi * x)
331     }
332
333     // argument reduction
334     z := Floor(x)
335     var n int
336     if z != x { // inexact
337         x = Mod(x, 2)
338         n = int(x * 4)
339     } else {
340         if x >= Two53 { // x must be even

```

```

341         x = 0
342         n = 0
343     } else {
344         if x < Two52 {
345             z = x + Two52 // exact
346         }
347         n = int(1 & Float64bits(z))
348         x = float64(n)
349         n <<= 2
350     }
351 }
352 switch n {
353 case 0:
354     x = Sin(Pi * x)
355 case 1, 2:
356     x = Cos(Pi * (0.5 - x))
357 case 3, 4:
358     x = Sin(Pi * (1 - x))
359 case 5, 6:
360     x = -Cos(Pi * (x - 1.5))
361 default:
362     x = Sin(Pi * (x - 2))
363 }
364 return -x
365 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/log.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Floating-point logarithm.
9 */
10
11 // The original C code, the long comment, and the constants
12 // below are from FreeBSD's /usr/src/lib/msun/src/e_log.c
13 // and came with this notice. The go code is a simpler
14 // version of the original C.
15 //
16 // =====
17 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
18 //
19 // Developed at SunPro, a Sun Microsystems, Inc. business.
20 // Permission to use, copy, modify, and distribute this
21 // software is freely granted, provided that this notice
22 // is preserved.
23 // =====
24 //
25 // __ieee754_log(x)
26 // Return the logarithm of x
27 //
28 // Method :
29 // 1. Argument Reduction: find k and f such that
30 //            $x = 2^{**k} * (1+f)$ ,
31 //           where  $\text{sqrt}(2)/2 < 1+f < \text{sqrt}(2)$  .
32 //
33 // 2. Approximation of log(1+f).
34 //   Let s = f/(2+f) ; based on  $\log(1+f) = \log(1+s) - \log(1-s)$ 
35 //            $= 2s + \frac{2}{3} s^{**3} + \frac{2}{5} s^{**5} + \dots$ ,
36 //            $= 2s + s*R$ 
37 // We use a special Reme algorithm on [0,0.1716] to generate
38 // a polynomial of degree 14 to approximate R. The maximum
39 // error of this polynomial approximation is bounded by 2**-55
40 // other words,
41 //
42 //           2       4       6       8       10
43 //           R(z) ~ L1*s +L2*s +L3*s +L4*s +L5*s +L6*s +L7*s
44 //           (the values of L1 to L7 are listed in the program) a
45 //           |         2         14         |         -58.45
```

```

45 //          | L1*s +...+L7*s      - R(z) | <= 2
46 //          |                               |
47 //      Note that 2s = f - s*f = f - hfsq + s*hfsq, where hf
48 //      In order to guarantee error in log below 1ulp, we co
49 //          log(1+f) = f - s*(f - R)          (if
50 //          log(1+f) = f - (hfsq - s*(hfsq+R)). (bet
51 //
52 //      3. Finally, log(x) = k*Ln2 + log(1+f).
53 //          = k*Ln2_hi+(f-(hfsq-(s*(hfsq+R))+
54 //      Here Ln2 is split into two floating point number:
55 //          Ln2_hi + Ln2_lo,
56 //      where n*Ln2_hi is always exact for |n| < 2000.
57 //
58 // Special cases:
59 //      log(x) is NaN with signal if x < 0 (including -INF)
60 //      log(+INF) is +INF; log(0) is -INF with signal;
61 //      log(NaN) is that NaN with no signal.
62 //
63 // Accuracy:
64 //      according to an error analysis, the error is always
65 //      1 ulp (unit in the last place).
66 //
67 // Constants:
68 // The hexadecimal values are the intended ones for the foll
69 // constants. The decimal values may be used, provided that
70 // compiler will convert from decimal to binary accurately e
71 // to produce the hexadecimal values shown.
72 //
73 // Log returns the natural logarithm of x.
74 //
75 // Special cases are:
76 //      Log(+Inf) = +Inf
77 //      Log(0) = -Inf
78 //      Log(x < 0) = NaN
79 //      Log(NaN) = NaN
80 func Log(x float64) float64
81
82 func log(x float64) float64 {
83     const (
84         Ln2Hi = 6.93147180369123816490e-01 /* 3fe62e
85         Ln2Lo = 1.90821492927058770002e-10 /* 3dea39
86         L1    = 6.6666666666666735130e-01  /* 3FE555
87         L2    = 3.999999999940941908e-01  /* 3FD999
88         L3    = 2.857142874366239149e-01  /* 3FD249
89         L4    = 2.222219843214978396e-01  /* 3FCC71
90         L5    = 1.818357216161805012e-01  /* 3FC746
91         L6    = 1.531383769920937332e-01  /* 3FC39A
92         L7    = 1.479819860511658591e-01  /* 3FC2F1
93     )
94

```

```

95         // special cases
96         switch {
97         case IsNaN(x) || IsInf(x, 1):
98             return x
99         case x < 0:
100            return NaN()
101         case x == 0:
102            return Inf(-1)
103         }
104
105         // reduce
106         f1, ki := Frexp(x)
107         if f1 < Sqrt2/2 {
108             f1 *= 2
109             ki--
110         }
111         f := f1 - 1
112         k := float64(ki)
113
114         // compute
115         s := f / (2 + f)
116         s2 := s * s
117         s4 := s2 * s2
118         t1 := s2 * (L1 + s4*(L3+s4*(L5+s4*L7)))
119         t2 := s4 * (L2 + s4*(L4+s4*L6))
120         R := t1 + t2
121         hfsq := 0.5 * f * f
122         return k*Ln2Hi - ((hfsq - (s*(hfsq+R) + k*Ln2Lo)) -
123     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/log10.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Log10 returns the decimal logarithm of x.
8 // The special cases are the same as for Log.
9 func Log10(x float64) float64
10
11 func log10(x float64) float64 {
12     return Log(x) * (1 / Ln10)
13 }
14
15 // Log2 returns the binary logarithm of x.
16 // The special cases are the same as for Log.
17 func Log2(x float64) float64
18
19 func log2(x float64) float64 {
20     return Log(x) * (1 / Ln2)
21 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/log1p.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // The original C code, the long comment, and the constants
8 // below are from FreeBSD's /usr/src/lib/msun/src/s_log1p.c
9 // and came with this notice. The go code is a simplified
10 // version of the original C.
11 //
12 // =====
13 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
14 //
15 // Developed at SunPro, a Sun Microsystems, Inc. business.
16 // Permission to use, copy, modify, and distribute this
17 // software is freely granted, provided that this notice
18 // is preserved.
19 // =====
20 //
21 //
22 // double log1p(double x)
23 //
24 // Method :
25 // 1. Argument Reduction: find k and f such that
26 //       $1+x = 2^{**k} * (1+f)$ ,
27 //      where  $\text{sqrt}(2)/2 < 1+f < \text{sqrt}(2)$  .
28 //
29 // Note. If k=0, then f=x is exact. However, if k!=0, t
30 // may not be representable exactly. In that case, a correction
31 // term is needed. Let u=1+x rounded. Let c = (1+x)-u, then
32 //  $\log(1+x) - \log(u) \sim c/u$ . Thus, we proceed to compute
33 // and add back the correction term c/u.
34 // (Note: when  $x > 2^{**53}$ , one can simply return  $\log(x)$ )
35 //
36 // 2. Approximation of  $\log_1p(f)$ .
37 // Let  $s = f/(2+f)$  ; based on  $\log(1+f) = \log(1+s) - \log(1-s)$ 
38 //       $= 2s + 2/3 s^{**3} + 2/5 s^{**5} + \dots$ ,
39 //       $= 2s + s^{**R}$ 
40 // We use a special Reme algorithm on  $[0,0.1716]$  to generate
41 // a polynomial of degree 14 to approximate R. The maximum error
42 // of this polynomial approximation is bounded by  $2^{**-5}$ .
43 // In other words,
44 //      2      4      6      8      10
```

```

45 //          R(z) ~ Lp1*s +Lp2*s +Lp3*s +Lp4*s +Lp5*s +Lp6*s
46 //          (the values of Lp1 to Lp7 are listed in the program)
47 //          and
48 //          |          2          14          |          -58.45
49 //          | Lp1*s +...+Lp7*s - R(z) | <= 2
50 //          |
51 //          Note that 2s = f - s*f = f - hfsq + s*hfsq, where hf
52 //          In order to guarantee error in log below 1ulp, we co
53 //          by
54 //          log1p(f) = f - (hfsq - s*(hfsq+R)).
55 //
56 //          3. Finally, log1p(x) = k*ln2 + log1p(f).
57 //          = k*ln2_hi+(f-(hfsq-(s*(hfsq+R))+k*
58 //          Here ln2 is split into two floating point number:
59 //          ln2_hi + ln2_lo,
60 //          where n*ln2_hi is always exact for |n| < 2000.
61 //
62 // Special cases:
63 //          log1p(x) is NaN with signal if x < -1 (including -IN
64 //          log1p(+INF) is +INF; log1p(-1) is -INF with signal;
65 //          log1p(NaN) is that NaN with no signal.
66 //
67 // Accuracy:
68 //          according to an error analysis, the error is always
69 //          1 ulp (unit in the last place).
70 //
71 // Constants:
72 //          The hexadecimal values are the intended ones for the foll
73 //          constants. The decimal values may be used, provided that
74 //          compiler will convert from decimal to binary accurately e
75 //          to produce the hexadecimal values shown.
76 //
77 // Note: Assuming log() return accurate answer, the followin
78 //          algorithm can be used to compute log1p(x) to within
79 //
80 //          u = 1+x;
81 //          if(u==1.0) return x ; else
82 //          return log(u)*(x/(u-1.0));
83 //
84 //          See HP-15C Advanced Functions Handbook, p.193.
85 //
86 // Log1p returns the natural logarithm of 1 plus its argumen
87 // It is more accurate than Log(1 + x) when x is near zero.
88 //
89 // Special cases are:
90 //          Log1p(+Inf) = +Inf
91 //          Log1p(±0) = ±0
92 //          Log1p(-1) = -Inf
93 //          Log1p(x < -1) = NaN
94 //          Log1p(NaN) = NaN

```

```

95 func Log1p(x float64) float64
96
97 func log1p(x float64) float64 {
98     const (
99         Sqrt2M1      = 4.142135623730950488017e-01 /
100        Sqrt2HalfM1 = -2.928932188134524755992e-01 /
101        Small       = 1.0 / (1 << 29)           /
102        Tiny        = 1.0 / (1 << 54)           /
103        Two53       = 1 << 53                   /
104        Ln2Hi       = 6.93147180369123816490e-01 /
105        Ln2Lo       = 1.90821492927058770002e-10 /
106        Lp1         = 6.6666666666666735130e-01 /
107        Lp2         = 3.999999999940941908e-01 /
108        Lp3         = 2.857142874366239149e-01 /
109        Lp4         = 2.222219843214978396e-01 /
110        Lp5         = 1.818357216161805012e-01 /
111        Lp6         = 1.531383769920937332e-01 /
112        Lp7         = 1.479819860511658591e-01 /
113    )
114
115    // special cases
116    switch {
117    case x < -1 || IsNaN(x): // includes -Inf
118        return NaN()
119    case x == -1:
120        return Inf(-1)
121    case IsInf(x, 1):
122        return Inf(1)
123    }
124
125    absx := x
126    if absx < 0 {
127        absx = -absx
128    }
129
130    var f float64
131    var iu uint64
132    k := 1
133    if absx < Sqrt2M1 { // |x| < Sqrt(2)-1
134        if absx < Small { // |x| < 2**-29
135            if absx < Tiny { // |x| < 2**-54
136                return x
137            }
138            return x - x*x*0.5
139        }
140        if x > Sqrt2HalfM1 { // Sqrt(2)/2-1 < x
141            // (Sqrt(2)/2-1) < x < (Sqrt(2)-1)
142            k = 0
143            f = x

```

```

144         iu = 1
145     }
146 }
147 var c float64
148 if k != 0 {
149     var u float64
150     if absx < Two53 { // 1<<53
151         u = 1.0 + x
152         iu = Float64bits(u)
153         k = int((iu >> 52) - 1023)
154         if k > 0 {
155             c = 1.0 - (u - x)
156         } else {
157             c = x - (u - 1.0) // correct
158             c /= u
159         }
160     } else {
161         u = x
162         iu = Float64bits(u)
163         k = int((iu >> 52) - 1023)
164         c = 0
165     }
166     iu &= 0x000fffffffffffff
167     if iu < 0x0006a09e667f3bcd { // mantissa of
168         u = Float64frombits(iu | 0x3ff000000)
169     } else {
170         k += 1
171         u = Float64frombits(iu | 0x3fe000000)
172         iu = (0x0010000000000000 - iu) >> 2
173     }
174     f = u - 1.0 // Sqrt(2)/2 < u < Sqrt(2)
175 }
176 hfsq := 0.5 * f * f
177 var s, R, z float64
178 if iu == 0 { // |f| < 2**-20
179     if f == 0 {
180         if k == 0 {
181             return 0
182         } else {
183             c += float64(k) * Ln2Lo
184             return float64(k)*Ln2Hi + c
185         }
186     }
187     R = hfsq * (1.0 - 0.6666666666666666*f) //
188     if k == 0 {
189         return f - R
190     }
191     return float64(k)*Ln2Hi - ((R - (float64(k)*
192

```

```

193         s = f / (2.0 + f)
194         z = s * s
195         R = z * (Lp1 + z*(Lp2+z*(Lp3+z*(Lp4+z*(Lp5+z*(Lp6+z*
196         if k == 0 {
197             return f - (hfsq - s*(hfsq+R))
198         }
199         return float64(k)*Ln2Hi - ((hfsq - (s*(hfsq+R) + (f1
200     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/logb.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Logb(x) returns the binary exponent of x.
8 //
9 // Special cases are:
10 //     Logb(±Inf) = ±Inf
11 //     Logb(0) = -Inf
12 //     Logb(NaN) = NaN
13 func Logb(x float64) float64 {
14     // special cases
15     switch {
16     case x == 0:
17         return Inf(-1)
18     case IsInf(x, 0):
19         return Inf(1)
20     case IsNaN(x):
21         return x
22     }
23     return float64(ilogb(x))
24 }
25
26 // Ilogb(x) returns the binary exponent of x as an integer.
27 //
28 // Special cases are:
29 //     Ilogb(±Inf) = MaxInt32
30 //     Ilogb(0) = MinInt32
31 //     Ilogb(NaN) = MaxInt32
32 func Ilogb(x float64) int {
33     // special cases
34     switch {
35     case x == 0:
36         return MinInt32
37     case IsNaN(x):
38         return MaxInt32
39     case IsInf(x, 0):
40         return MaxInt32
41     }
42     return ilogb(x)
43 }
44
```

```
45 // logb returns the binary exponent of x. It assumes x is fi
46 // non-zero.
47 func ilogb(x float64) int {
48     x, exp := normalize(x)
49     return int((Float64bits(x)>>shift)&mask) - bias + ex
50 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/mod.go

```
1 // Copyright 2009-2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Floating-point mod function.
9 */
10
11 // Mod returns the floating-point remainder of x/y.
12 // The magnitude of the result is less than y and its
13 // sign agrees with that of x.
14 //
15 // Special cases are:
16 //     Mod( $\pm$ Inf, y) = NaN
17 //     Mod(NaN, y) = NaN
18 //     Mod(x, 0) = NaN
19 //     Mod(x,  $\pm$ Inf) = x
20 //     Mod(x, NaN) = NaN
21 func Mod(x, y float64) float64
22
23 func mod(x, y float64) float64 {
24     if y == 0 || IsInf(x, 0) || IsNaN(x) || IsNaN(y) {
25         return NaN()
26     }
27     if y < 0 {
28         y = -y
29     }
30
31     yfr, yexp := Frexp(y)
32     sign := false
33     r := x
34     if x < 0 {
35         r = -x
36         sign = true
37     }
38
39     for r >= y {
40         rfr, rexp := Frexp(r)
41         if rfr < yfr {
42             rexp = rexp - 1
43         }
44         r = r - Ldexp(y, rexp-yexp)
```

```
45     }
46     if sign {
47         r = -r
48     }
49     return r
50 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/modf.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Modf returns integer and fractional floating-point number
8 // that sum to f. Both values have the same sign as f.
9 //
10 // Special cases are:
11 //     Modf(±Inf) = ±Inf, NaN
12 //     Modf(NaN) = NaN, NaN
13 func Modf(f float64) (int float64, frac float64)
14
15 func modf(f float64) (int float64, frac float64) {
16     if f < 1 {
17         if f < 0 {
18             int, frac = Modf(-f)
19             return -int, -frac
20         }
21         return 0, f
22     }
23
24     x := Float64bits(f)
25     e := uint(x>>shift)&mask - bias
26
27     // Keep the top 12+e bits, the integer part; clear t
28     if e < 64-12 {
29         x &^= 1<<(64-12-e) - 1
30     }
31     int = Float64frombits(x)
32     frac = f - int
33     return
34 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/nextafter.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Nextafter returns the next representable value after x to
8 // If x == y, then x is returned.
9 //
10 // Special cases are:
11 //     Nextafter(NaN, y) = NaN
12 //     Nextafter(x, NaN) = NaN
13 func Nextafter(x, y float64) (r float64) {
14     switch {
15     case IsNaN(x) || IsNaN(y): // special case
16         r = NaN()
17     case x == y:
18         r = x
19     case x == 0:
20         r = Copysign(Float64frombits(1), y)
21     case (y > x) == (x > 0):
22         r = Float64frombits(Float64bits(x) + 1)
23     default:
24         r = Float64frombits(Float64bits(x) - 1)
25     }
26     return
27 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/pow.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 func isOddInt(x float64) bool {
8     xi, xf := Modf(x)
9     return xf == 0 && int64(xi)&1 == 1
10 }
11
12 // Special cases taken from FreeBSD's /usr/src/lib/msun/src/
13 // updated by IEEE Std. 754-2008 "Section 9.2.1 Special valu
14
15 // Pow returns x**y, the base-x exponential of y.
16 //
17 // Special cases are (in order):
18 //     Pow(x, ±0) = 1 for any x
19 //     Pow(1, y) = 1 for any y
20 //     Pow(x, 1) = x for any x
21 //     Pow(NaN, y) = NaN
22 //     Pow(x, NaN) = NaN
23 //     Pow(±0, y) = ±Inf for y an odd integer < 0
24 //     Pow(±0, -Inf) = +Inf
25 //     Pow(±0, +Inf) = +0
26 //     Pow(±0, y) = +Inf for finite y < 0 and not an odd in
27 //     Pow(±0, y) = ±0 for y an odd integer > 0
28 //     Pow(±0, y) = +0 for finite y > 0 and not an odd inte
29 //     Pow(-1, ±Inf) = 1
30 //     Pow(x, +Inf) = +Inf for |x| > 1
31 //     Pow(x, -Inf) = +0 for |x| > 1
32 //     Pow(x, +Inf) = +0 for |x| < 1
33 //     Pow(x, -Inf) = +Inf for |x| < 1
34 //     Pow(+Inf, y) = +Inf for y > 0
35 //     Pow(+Inf, y) = +0 for y < 0
36 //     Pow(-Inf, y) = Pow(-0, -y)
37 //     Pow(x, y) = NaN for finite x < 0 and finite non-inte
38 func Pow(x, y float64) float64 {
39     switch {
40     case y == 0 || x == 1:
41         return 1
42     case y == 1:
43         return x
44     case y == 0.5:
```

```

45         return Sqrt(x)
46     case y == -0.5:
47         return 1 / Sqrt(x)
48     case IsNaN(x) || IsNaN(y):
49         return NaN()
50     case x == 0:
51         switch {
52             case y < 0:
53                 if isOddInt(y) {
54                     return Copysign(Inf(1), x)
55                 }
56                 return Inf(1)
57             case y > 0:
58                 if isOddInt(y) {
59                     return x
60                 }
61                 return 0
62         }
63     case IsInf(y, 0):
64         switch {
65             case x == -1:
66                 return 1
67             case (Abs(x) < 1) == IsInf(y, 1):
68                 return 0
69             default:
70                 return Inf(1)
71         }
72     case IsInf(x, 0):
73         if IsInf(x, -1) {
74             return Pow(1/x, -y) // Pow(-0, -y)
75         }
76         switch {
77             case y < 0:
78                 return 0
79             case y > 0:
80                 return Inf(1)
81         }
82     }
83
84     absy := y
85     flip := false
86     if absy < 0 {
87         absy = -absy
88         flip = true
89     }
90     yi, yf := Modf(absy)
91     if yf != 0 && x < 0 {
92         return NaN()
93     }
94     if yi >= 1<<63 {

```

```

95         return Exp(y * Log(x))
96     }
97
98     // ans = a1 * 2**ae (= 1 for now).
99     a1 := 1.0
100    ae := 0
101
102    // ans *= x**yf
103    if yf != 0 {
104        if yf > 0.5 {
105            yf--
106            yi++
107        }
108        a1 = Exp(yf * Log(x))
109    }
110
111    // ans *= x**yi
112    // by multiplying in successive squarings
113    // of x according to bits of yi.
114    // accumulate powers of two into exp.
115    x1, xe := Frexp(x)
116    for i := int64(yi); i != 0; i >>= 1 {
117        if i&1 == 1 {
118            a1 *= x1
119            ae += xe
120        }
121        x1 *= x1
122        xe <<= 1
123        if x1 < .5 {
124            x1 += x1
125            xe--
126        }
127    }
128
129    // ans = a1*2**ae
130    // if flip { ans = 1 / ans }
131    // but in the opposite order
132    if flip {
133        a1 = 1 / a1
134        ae = -ae
135    }
136    return Ldexp(a1, ae)
137 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/pow10.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // This table might overflow 127-bit exponent representation
8 // In that case, truncate it after 1.0e38.
9 var pow10tab [70]float64
10
11 // Pow10 returns 10**e, the base-10 exponential of e.
12 //
13 // Special cases are:
14 //     Pow10(e) = +Inf for e > 309
15 //     Pow10(e) = 0 for e < -324
16 func Pow10(e int) float64 {
17     if e <= -325 {
18         return 0
19     } else if e > 309 {
20         return Inf(1)
21     }
22
23     if e < 0 {
24         return 1 / Pow10(-e)
25     }
26     if e < len(pow10tab) {
27         return pow10tab[e]
28     }
29     m := e / 2
30     return Pow10(m) * Pow10(e-m)
31 }
32
33 func init() {
34     pow10tab[0] = 1.0e0
35     pow10tab[1] = 1.0e1
36     for i := 2; i < len(pow10tab); i++ {
37         m := i / 2
38         pow10tab[i] = pow10tab[m] * pow10tab[i-m]
39     }
40 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/remainder.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // The original C code and the the comment below are from
8 // FreeBSD's /usr/src/lib/msun/src/e_remainder.c and came
9 // with this notice. The go code is a simplified version of
10 // the original C.
11 //
12 // =====
13 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights r
14 //
15 // Developed at SunPro, a Sun Microsystems, Inc. business.
16 // Permission to use, copy, modify, and distribute this
17 // software is freely granted, provided that this notice
18 // is preserved.
19 // =====
20 //
21 // __ieee754_remainder(x,y)
22 // Return :
23 //     returns x REM y = x - [x/y]*y as if in infinite
24 //     precision arithmetic, where [x/y] is the (infinite b
25 //     integer nearest x/y (in half way cases, choose the e
26 // Method :
27 //     Based on Mod() returning x - [x/y]chopped * y exac
28
29 // Remainder returns the IEEE 754 floating-point remainder o
30 //
31 // Special cases are:
32 //     Remainder(±Inf, y) = NaN
33 //     Remainder(NaN, y) = NaN
34 //     Remainder(x, 0) = NaN
35 //     Remainder(x, ±Inf) = x
36 //     Remainder(x, NaN) = NaN
37 func Remainder(x, y float64) float64
38
39 func remainder(x, y float64) float64 {
40     const (
41         Tiny      = 4.45014771701440276618e-308 // 0xC
```

```

42             HalfMax = MaxFloat64 / 2
43         )
44     // special cases
45     switch {
46     case IsNaN(x) || IsNaN(y) || IsInf(x, 0) || y == 0:
47         return NaN()
48     case IsInf(y, 0):
49         return x
50     }
51     sign := false
52     if x < 0 {
53         x = -x
54         sign = true
55     }
56     if y < 0 {
57         y = -y
58     }
59     if x == y {
60         return 0
61     }
62     if y <= HalfMax {
63         x = Mod(x, y+y) // now x < 2y
64     }
65     if y < Tiny {
66         if x+x > y {
67             x -= y
68             if x+x >= y {
69                 x -= y
70             }
71         }
72     } else {
73         yHalf := 0.5 * y
74         if x > yHalf {
75             x -= y
76             if x >= yHalf {
77                 x -= y
78             }
79         }
80     }
81     if sign {
82         x = -x
83     }
84     return x
85 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/signbit.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.  
2 // Use of this source code is governed by a BSD-style  
3 // license that can be found in the LICENSE file.  
4  
5 package math  
6  
7 // Signbit returns true if x is negative or negative zero.  
8 func Signbit(x float64) bool {  
9     return Float64bits(x)&(1<<63) != 0  
10 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/sin.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Floating-point sine and cosine.
9 */
10
11 // The original C code, the long comment, and the constants
12 // below were from http://netlib.sandia.gov/cephes/cmath/sin
13 // available from http://www.netlib.org/cephes/cmath.tgz.
14 // The go code is a simplified version of the original C.
15 //
16 //     sin.c
17 //
18 //     Circular sine
19 //
20 // SYNOPSIS:
21 //
22 // double x, y, sin();
23 // y = sin( x );
24 //
25 // DESCRIPTION:
26 //
27 // Range reduction is into intervals of pi/4. The reduction
28 // eliminated by contriving an extended precision modular ar
29 //
30 // Two polynomial approximating functions are employed.
31 // Between 0 and pi/4 the sine is approximated by
32 //     x + x**3 P(x**2).
33 // Between pi/4 and pi/2 the cosine is represented as
34 //     1 - x**2 Q(x**2).
35 //
36 // ACCURACY:
37 //
38 //                                     Relative error:
39 // arithmetic    domain    # trials    peak        rms
40 //     DEC        0, 10     150000    3.0e-17     7.8e-18
41 //     IEEE -1.07e9,+1.07e9  130000    2.1e-16     5.4e-17
42 //
43 // Partial loss of accuracy begins to occur at x = 2**30 = 1
44 // is not gradual, but jumps suddenly to about 1 part in 10e
```

```

45 // be meaningless for x > 2**49 = 5.6e14.
46 //
47 //     cos.c
48 //
49 //     Circular cosine
50 //
51 // SYNOPSIS:
52 //
53 // double x, y, cos();
54 // y = cos( x );
55 //
56 // DESCRIPTION:
57 //
58 // Range reduction is into intervals of pi/4. The reduction
59 // eliminated by contriving an extended precision modular ar
60 //
61 // Two polynomial approximating functions are employed.
62 // Between 0 and pi/4 the cosine is approximated by
63 //     1 - x**2 Q(x**2).
64 // Between pi/4 and pi/2 the sine is represented as
65 //     x + x**3 P(x**2).
66 //
67 // ACCURACY:
68 //
69 //                                     Relative error:
70 // arithmetic   domain      # trials      peak          rms
71 //   IEEE -1.07e9,+1.07e9  130000      2.1e-16      5.4e-17
72 //   DEC        0,+1.07e9   17000       3.0e-17      7.2e-18
73 //
74 // Cephes Math Library Release 2.8:  June, 2000
75 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
76 //
77 // The readme file at http://netlib.sandia.gov/cephes/ says:
78 //     Some software in this archive may be from the book _Me
79 // Programs for Mathematical Functions_ (Prentice-Hall or Si
80 // International, 1989) or from the Cephes Mathematical Libr
81 // commercial product. In either event, it is copyrighted by
82 // What you see here may be used freely but it comes with no
83 // guarantee.
84 //
85 //     The two known misprints in the book are repaired here i
86 // source listings for the gamma function and the incomplete
87 // integral.
88 //
89 //     Stephen L. Moshier
90 //     moshier@na-net.ornl.gov
91
92 // sin coefficients
93 var _sin = [...]float64{
94     1.58962301576546568060E-10, // 0x3de5d8fd1fd19ccd

```

```

95         -2.50507477628578072866E-8, // 0xbe5ae5e5a9291f5d
96         2.75573136213857245213E-6, // 0x3ec71de3567d48a1
97         -1.98412698295895385996E-4, // 0xbf2a01a019bdfd03
98         8.3333333332211858878E-3, // 0x3f8111111110f7d0
99         -1.66666666666666307295E-1, // 0xbfc555555555548
100    }
101
102    // cos coefficients
103    var _cos = [...]float64{
104        -1.13585365213876817300E-11, // 0xbda8fa49a0861a9b
105        2.08757008419747316778E-9, // 0x3e21ee9d7b4e3f05
106        -2.75573141792967388112E-7, // 0xbe927e4f7eac4bc6
107        2.48015872888517045348E-5, // 0x3efa01a019c844f5
108        -1.3888888888730564116E-3, // 0xbf56c16c16c14f91
109        4.166666666665929218E-2, // 0x3fa55555555554b
110    }
111
112    // Cos returns the cosine of x.
113    //
114    // Special cases are:
115    //     Cos( $\pm$ Inf) = NaN
116    //     Cos(NaN) = NaN
117    func Cos(x float64) float64
118
119    func cos(x float64) float64 {
120        const (
121            PI4A = 7.85398125648498535156E-1
122            PI4B = 3.77489470793079817668E-8
123            PI4C = 2.69515142907905952645E-15
124            M4PI = 1.27323954473516254282117188267875462
125        )
126        // special cases
127        switch {
128        case IsNaN(x) || IsInf(x, 0):
129            return NaN()
130        }
131
132        // make argument positive
133        sign := false
134        if x < 0 {
135            x = -x
136        }
137
138        j := int64(x * M4PI) // integer part of x/(Pi/4), as
139        y := float64(j) // integer part of x/(Pi/4), as
140
141        // map zeros to origin
142        if j&1 == 1 {
143            j += 1

```

```

144         y += 1
145     }
146     j &= 7 // octant modulo 2Pi radians (360 degrees)
147     if j > 3 {
148         j -= 4
149         sign = !sign
150     }
151     if j > 1 {
152         sign = !sign
153     }
154
155     z := ((x - y*PI4A) - y*PI4B) - y*PI4C // Extended pr
156     zz := z * z
157     if j == 1 || j == 2 {
158         y = z + z*zz*(((((_sin[0]*zz)+_sin[1]))*zz+_
159     } else {
160         y = 1.0 - 0.5*zz + zz*zz*(((((_cos[0]*zz)+_
161     }
162     if sign {
163         y = -y
164     }
165     return y
166 }
167
168 // Sin returns the sine of x.
169 //
170 // Special cases are:
171 //     Sin(±0) = ±0
172 //     Sin(±Inf) = NaN
173 //     Sin(NaN) = NaN
174 func Sin(x float64) float64
175
176 func sin(x float64) float64 {
177     const (
178         PI4A = 7.85398125648498535156E-1
179         PI4B = 3.77489470793079817668E-8
180         PI4C = 2.69515142907905952645E-15
181         M4PI = 1.27323954473516254282117188267875462
182     )
183     // special cases
184     switch {
185     case x == 0 || IsNaN(x):
186         return x // return ±0 || NaN()
187     case IsInf(x, 0):
188         return NaN()
189     }
190
191     // make argument positive but save the sign
192     sign := false

```

```

193     if x < 0 {
194         x = -x
195         sign = true
196     }
197
198     j := int64(x * M4PI) // integer part of x/(Pi/4), as
199     y := float64(j)      // integer part of x/(Pi/4), as
200
201     // map zeros to origin
202     if j&1 == 1 {
203         j += 1
204         y += 1
205     }
206     j &= 7 // octant modulo 2Pi radians (360 degrees)
207     // reflect in x axis
208     if j > 3 {
209         sign = !sign
210         j -= 4
211     }
212
213     z := ((x - y*PI4A) - y*PI4B) - y*PI4C // Extended pr
214     zz := z * z
215     if j == 1 || j == 2 {
216         y = 1.0 - 0.5*zz + zz*zz*((((((_cos[0]*zz)+_
217     } else {
218         y = z + z*zz*((((((_sin[0]*zz)+_sin[1]))*zz+_
219     }
220     if sign {
221         y = -y
222     }
223     return y
224 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/sincos.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Coefficients _sin[] and _cos[] are found in pkg/math/sin.
8
9 // Sincos(x) returns Sin(x), Cos(x).
10 //
11 // Special cases are:
12 //     Sincos( $\pm 0$ ) =  $\pm 0$ , 1
13 //     Sincos( $\pm \text{Inf}$ ) = NaN, NaN
14 //     Sincos(NaN) = NaN, NaN
15 func Sincos(x float64) (sin, cos float64)
16
17 func sincos(x float64) (sin, cos float64) {
18     const (
19         PI4A = 7.85398125648498535156E-1
20         PI4B = 3.77489470793079817668E-8
21         PI4C = 2.69515142907905952645E-15
22         M4PI = 1.27323954473516254282117188267875462
23     )
24     // special cases
25     switch {
26     case x == 0:
27         return x, 1 // return  $\pm 0.0$ , 1.0
28     case IsNaN(x) || IsInf(x, 0):
29         return NaN(), NaN()
30     }
31
32     // make argument positive
33     sinSign, cosSign := false, false
34     if x < 0 {
35         x = -x
36         sinSign = true
37     }
38
39     j := int64(x * M4PI) // integer part of  $x/(Pi/4)$ , as
40     y := float64(j)     // integer part of  $x/(Pi/4)$ , as
41
42     if j&1 == 1 { // map zeros to origin
43         j += 1
44         y += 1
```

```

45     }
46     j &= 7      // octant modulo 2Pi radians (360 degrees
47     if j > 3 { // reflect in x axis
48         j -= 4
49         sinSign, cosSign = !sinSign, !cosSign
50     }
51     if j > 1 {
52         cosSign = !cosSign
53     }
54
55     z := ((x - y*PI4A) - y*PI4B) - y*PI4C // Extended pr
56     zz := z * z
57     cos = 1.0 - 0.5*zz + zz*zz*(((((_cos[0]*zz)+_cos[1]
58     sin = z + z*zz*(((((_sin[0]*zz)+_sin[1])*zz+_sin[2]
59     if j == 1 || j == 2 {
60         sin, cos = cos, sin
61     }
62     if cosSign {
63         cos = -cos
64     }
65     if sinSign {
66         sin = -sin
67     }
68     return
69 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/sinh.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Floating-point hyperbolic sine and cosine.
9
10    The exponential func is called for arguments
11    greater in magnitude than 0.5.
12
13    A series is used for arguments smaller in magnitude
14
15    Cosh(x) is computed from the exponential func for
16    all arguments.
17 */
18
19 // Sinh returns the hyperbolic sine of x.
20 //
21 // Special cases are:
22 //     Sinh(±0) = ±0
23 //     Sinh(±Inf) = ±Inf
24 //     Sinh(NaN) = NaN
25 func Sinh(x float64) float64 {
26     // The coefficients are #2029 from Hart & Cheney. (2
27     const (
28         P0 = -0.6307673640497716991184787251e+6
29         P1 = -0.8991272022039509355398013511e+5
30         P2 = -0.2894211355989563807284660366e+4
31         P3 = -0.2630563213397497062819489e+2
32         Q0 = -0.6307673640497716991212077277e+6
33         Q1 = 0.1521517378790019070696485176e+5
34         Q2 = -0.173678953558233699533450911e+3
35     )
36
37     sign := false
38     if x < 0 {
39         x = -x
40         sign = true
41     }
42
43     var temp float64
44     switch true {
```

```

45         case x > 21:
46             temp = Exp(x) / 2
47
48         case x > 0.5:
49             temp = (Exp(x) - Exp(-x)) / 2
50
51         default:
52             sq := x * x
53             temp = (((P3*sq+P2)*sq+P1)*sq + P0) * x
54             temp = temp / (((sq+Q2)*sq+Q1)*sq + Q0)
55         }
56
57         if sign {
58             temp = -temp
59         }
60         return temp
61     }
62
63     // Cosh returns the hyperbolic cosine of x.
64     //
65     // Special cases are:
66     //     Cosh(±0) = 1
67     //     Cosh(±Inf) = +Inf
68     //     Cosh(NaN) = NaN
69     func Cosh(x float64) float64 {
70         if x < 0 {
71             x = -x
72         }
73         if x > 21 {
74             return Exp(x) / 2
75         }
76         return (Exp(x) + Exp(-x)) / 2
77     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/sqrt.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 // Sqrt returns the square root of x.
8 //
9 // Special cases are:
10 //     Sqrt(+Inf) = +Inf
11 //     Sqrt(±0) = ±0
12 //     Sqrt(x < 0) = NaN
13 //     Sqrt(NaN) = NaN
14 func Sqrt(x float64) float64
15
16 // The original C code and the long comment below are
17 // from FreeBSD's /usr/src/lib/msun/src/e_sqrt.c and
18 // came with this notice. The go code is a simplified
19 // version of the original C.
20 //
21 // =====
22 // Copyright (C) 1993 by Sun Microsystems, Inc. All rights reserved.
23 //
24 // Developed at SunPro, a Sun Microsystems, Inc. business.
25 // Permission to use, copy, modify, and distribute this
26 // software is freely granted, provided that this notice
27 // is preserved.
28 // =====
29 //
30 // __ieee754_sqrt(x)
31 // Return correctly rounded sqrt.
32 //
33 // | Use the hardware sqrt if you have one |
34 // -----
35 // Method:
36 //   Bit by bit method using integer arithmetic. (Slow, but
37 //   1. Normalization
38 //     Scale x to y in [1,4) with even powers of 2:
39 //     find an integer k such that 1 <= (y=x*2**(2k)) < 4,
40 //     sqrt(x) = 2**k * sqrt(y)
41 //   2. Bit by bit computation
42 //     Let q = sqrt(y) truncated to i bit after binary point
43 //     i
44 //     i+1     2
```

```

45 //          si = 2*qi , and          yi = 2 * ( yi - qi ).
46 //          i          i          i          i
47 //
48 //      To compute qi+1 from qi , one checks whether
49 //
50 //
51 //          -(i+1) 2
52 //          (qi + 2 ) <= y.
53 //          i
54 //
55 //      If (2) is false, then qi+1 = qi ; otherwise qi+1 = qi +
56 //          i+1    i          i+1    i
57 //
58 //      With some algebraic manipulation, it is not difficult
59 //      that (2) is equivalent to
60 //          -(i+1)
61 //          si + 2 <= yi
62 //          i          i
63 //
64 //      The advantage of (3) is that si and yi can be computed
65 //          i          i
66 //      the following recurrence formula:
67 //      if (3) is false
68 //
69 //          si+1 = si ,          yi+1 = yi ;
70 //          i+1    i          i+1    i
71 //
72 //      otherwise,
73 //          -i          -(i+1)
74 //          si+1 = si + 2 ,          yi+1 = yi - si - 2
75 //          i+1    i          i+1    i          i
76 //
77 //      One may easily use induction to prove (4) and (5).
78 //      Note. Since the left hand side of (3) contain only i
79 //      it does not necessary to do a full (53-bit) computation
80 //      in (3).
81 //      3. Final rounding
82 //      After generating the 53 bits result, we compute one
83 //      Together with the remainder, we can decide whether the
84 //      result is exact, bigger than 1/2ulp, or less than 1/
85 //      (it will never equal to 1/2ulp).
86 //      The rounding mode can be detected by checking whether
87 //      huge + tiny is equal to huge, and whether huge - tiny
88 //      equal to huge for some floating point number "huge"
89 //
90 //
91 //      Notes: Rounding mode detection omitted. The constants "
92 //      and "bias" are found in src/pkg/math/bits.go
93 //
94 //      Sqrt returns the square root of x.

```

```

95 //
96 // Special cases are:
97 //     Sqrt(+Inf) = +Inf
98 //     Sqrt(±0) = ±0
99 //     Sqrt(x < 0) = NaN
100 //     Sqrt(NaN) = NaN
101 func sqrt(x float64) float64 {
102     // special cases
103     switch {
104     case x == 0 || IsNaN(x) || IsInf(x, 1):
105         return x
106     case x < 0:
107         return NaN()
108     }
109     ix := Float64bits(x)
110     // normalize x
111     exp := int((ix >> shift) & mask)
112     if exp == 0 { // subnormal x
113         for ix&1<<shift == 0 {
114             ix <<= 1
115             exp--
116         }
117         exp++
118     }
119     exp -= bias // unbiased exponent
120     ix &^= mask << shift
121     ix |= 1 << shift
122     if exp&1 == 1 { // odd exp, double x to make it even
123         ix <<= 1
124     }
125     exp >>= 1 // exp = exp/2, exponent of square root
126     // generate sqrt(x) bit by bit
127     ix <<= 1
128     var q, s uint64 // q = sqrt(x)
129     r := uint64(1 << (shift + 1)) // r = moving bit from
130     for r != 0 {
131         t := s + r
132         if t <= ix {
133             s = t + r
134             ix -= t
135             q += r
136         }
137         ix <<= 1
138         r >>= 1
139     }
140     // final rounding
141     if ix != 0 { // remainder, result not exact
142         q += q & 1 // round according to extra bit
143     }

```

```
144         ix = q>>1 + uint64(exp-1+bias)<<shift // significand
145         return Float64frombits(ix)
146     }
147
148     func sqrtC(f float64, r *float64) {
149         *r = sqrt(f)
150     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/tan.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Floating-point tangent.
9 */
10
11 // The original C code, the long comment, and the constants
12 // below were from http://netlib.sandia.gov/cephes/cmath/sin
13 // available from http://www.netlib.org/cephes/cmath.tgz.
14 // The go code is a simplified version of the original C.
15 //
16 //     tan.c
17 //
18 //     Circular tangent
19 //
20 // SYNOPSIS:
21 //
22 // double x, y, tan();
23 // y = tan( x );
24 //
25 // DESCRIPTION:
26 //
27 // Returns the circular tangent of the radian argument x.
28 //
29 // Range reduction is modulo pi/4. A rational function
30 //  $x + x^3 P(x^2)/Q(x^2)$ 
31 // is employed in the basic interval  $[0, \pi/4]$ .
32 //
33 // ACCURACY:
34 //
35 //                                     Relative error:
36 // arithmetic    domain    # trials    peak       rms
37 //   DEC         +-1.07e9    44000     4.1e-17    1.0e-17
38 //   IEEE        +-1.07e9    30000     2.9e-16    8.1e-17
39 //
40 // Partial loss of accuracy begins to occur at  $x = 2^{30} = 1$ 
41 // is not gradual, but jumps suddenly to about 1 part in  $10^e$ 
42 // be meaningless for  $x > 2^{49} = 5.6e14$ .
43 // [Accuracy loss statement from sin.go comments.]
44 //
45 // Cephes Math Library Release 2.8:  June, 2000
```

```

45 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
46 //
47 // The readme file at http://netlib.sandia.gov/cephes/ says:
48 //     Some software in this archive may be from the book _Me
49 // Programs for Mathematical Functions_ (Prentice-Hall or Si
50 // International, 1989) or from the Cephes Mathematical Libr
51 // commercial product. In either event, it is copyrighted by
52 // What you see here may be used freely but it comes with no
53 // guarantee.
54 //
55 //     The two known misprints in the book are repaired here i
56 // source listings for the gamma function and the incomplete
57 // integral.
58 //
59 //     Stephen L. Moshier
60 //     moshier@na-net.ornl.gov
61
62 // tan coefficients
63 var _tanP = [...]float64{
64     -1.30936939181383777646E4, // 0xc0c992d8d24f3f38
65     1.15351664838587416140E6, // 0x413199eca5fc9ddd
66     -1.79565251976484877988E7, // 0xc1711fead3299176
67 }
68 var _tanQ = [...]float64{
69     1.00000000000000000000E0,
70     1.36812963470692954678E4, //0x40cab8a5eeb36572
71     -1.32089234440210967447E6, //0xc13427bc582abc96
72     2.50083801823357915839E7, //0x4177d98fc2ead8ef
73     -5.38695755929454629881E7, //0xc189afe03cbe5a31
74 }
75
76 // Tan returns the tangent of x.
77 //
78 // Special cases are:
79 //     Tan( $\pm 0$ ) =  $\pm 0$ 
80 //     Tan( $\pm \text{Inf}$ ) = NaN
81 //     Tan(NaN) = NaN
82 func Tan(x float64) float64
83
84 func tan(x float64) float64 {
85     const (
86         PI4A = 7.85398125648498535156E-1
87         PI4B = 3.77489470793079817668E-8
88         PI4C = 2.69515142907905952645E-15
89         M4PI = 1.27323954473516254282117188267875462
90     )
91     // special cases
92     switch {
93     case x == 0 || IsNaN(x):
94         return x // return  $\pm 0$  || NaN()

```

```

95     case IsInf(x, 0):
96         return NaN()
97     }
98
99     // make argument positive but save the sign
100    sign := false
101    if x < 0 {
102        x = -x
103        sign = true
104    }
105
106    j := int64(x * M4PI) // integer part of x/(Pi/4), as
107    y := float64(j)      // integer part of x/(Pi/4), as
108
109    /* map zeros and singularities to origin */
110    if j&1 == 1 {
111        j += 1
112        y += 1
113    }
114
115    z := ((x - y*PI4A) - y*PI4B) - y*PI4C
116    zz := z * z
117
118    if zz > 1e-14 {
119        y = z + z*(zz*(((tanP[0]*zz)+tanP[1])*zz+_
120    } else {
121        y = z
122    }
123    if j&2 == 2 {
124        y = -1 / y
125    }
126    if sign {
127        y = -y
128    }
129    return y
130 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/tanh.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 /*
8     Floating-point hyperbolic tangent.
9
10    Sinh and Cosh are called except for large arguments,
11    would cause overflow improperly.
12 */
13
14 // Tanh computes the hyperbolic tangent of x.
15 //
16 // Special cases are:
17 //     Tanh(±0) = ±0
18 //     Tanh(±Inf) = ±1
19 //     Tanh(NaN) = NaN
20 func Tanh(x float64) float64 {
21     if x < 0 {
22         x = -x
23         if x > 21 {
24             return -1
25         }
26         return -Sinh(x) / Cosh(x)
27     }
28     if x > 21 {
29         return 1
30     }
31     return Sinh(x) / Cosh(x)
32 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/unsafe.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package math
6
7 import "unsafe"
8
9 // Float32bits returns the IEEE 754 binary representation of
10 func Float32bits(f float32) uint32 { return *(*uint32)(unsafe.Pointer(&f)) }
11
12 // Float32frombits returns the floating point number corresponding
13 // to the IEEE 754 binary representation b.
14 func Float32frombits(b uint32) float32 { return *(*float32)(unsafe.Pointer(&b)) }
15
16 // Float64bits returns the IEEE 754 binary representation of
17 func Float64bits(f float64) uint64 { return *(*uint64)(unsafe.Pointer(&f)) }
18
19 // Float64frombits returns the floating point number corresponding
20 // to the IEEE 754 binary representation b.
21 func Float64frombits(b uint64) float64 { return *(*float64)(unsafe.Pointer(&b)) }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/big/arith.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file provides Go implementations of elementary multi
6 // arithmetic operations on word vectors. Needed for platfor
7 // assembly implementations of these routines.
8
9 package big
10
11 // A Word represents a single digit of a multi-precision uns
12 type Word uintptr
13
14 const (
15     // Compute the size _S of a Word in bytes.
16     _m      = ^Word(0)
17     _logS   = _m>>8&1 + _m>>16&1 + _m>>32&1
18     _S      = 1 << _logS
19
20     _W = _S << 3 // word size in bits
21     _B = 1 << _W // digit base
22     _M = _B - 1 // digit mask
23
24     _W2 = _W / 2 // half word size in bits
25     _B2 = 1 << _W2 // half digit base
26     _M2 = _B2 - 1 // half digit mask
27 )
28
29 // -----
30 // Elementary operations on words
31 //
32 // These operations are used by the vector operations below.
33
34 // z1<<_W + z0 = x+y+c, with c == 0 or 1
35 func addWW_g(x, y, c Word) (z1, z0 Word) {
36     yc := y + c
37     z0 = x + yc
38     if z0 < x || yc < y {
39         z1 = 1
40     }
41     return
42 }
43
44 // z1<<_W + z0 = x-y-c, with c == 0 or 1
```

```

45 func subWW_g(x, y, c Word) (z1, z0 Word) {
46     yc := y + c
47     z0 = x - yc
48     if z0 > x || yc < y {
49         z1 = 1
50     }
51     return
52 }
53
54 // z1<<_W + z0 = x*y
55 // Adapted from Warren, Hacker's Delight, p. 132.
56 func mulWW_g(x, y Word) (z1, z0 Word) {
57     x0 := x & _M2
58     x1 := x >> _W2
59     y0 := y & _M2
60     y1 := y >> _W2
61     w0 := x0 * y0
62     t := x1*y0 + w0>>_W2
63     w1 := t & _M2
64     w2 := t >> _W2
65     w1 += x0 * y1
66     z1 = x1*y1 + w2 + w1>>_W2
67     z0 = x * y
68     return
69 }
70
71 // z1<<_W + z0 = x*y + c
72 func mulAddWW_g(x, y, c Word) (z1, z0 Word) {
73     z1, zz0 := mulWW(x, y)
74     if z0 = zz0 + c; z0 < zz0 {
75         z1++
76     }
77     return
78 }
79
80 // Length of x in bits.
81 func bitLen_g(x Word) (n int) {
82     for ; x >= 0x8000; x >>= 16 {
83         n += 16
84     }
85     if x >= 0x80 {
86         x >>= 8
87         n += 8
88     }
89     if x >= 0x8 {
90         x >>= 4
91         n += 4
92     }
93     if x >= 0x2 {
94         x >>= 2

```

```

95             n += 2
96         }
97         if x >= 0x1 {
98             n++
99         }
100        return
101    }
102
103    // log2 computes the integer binary logarithm of x.
104    // The result is the integer n for which  $2^n \leq x < 2^{(n+1)}$ .
105    // If  $x == 0$ , the result is -1.
106    func log2(x Word) int {
107        return bitLen(x) - 1
108    }
109
110    // Number of leading zeros in x.
111    func leadingZeros(x Word) uint {
112        return uint(_W - bitLen(x))
113    }
114
115    //  $q = (u1 \ll _W + u0 - r) / y$ 
116    // Adapted from Warren, Hacker's Delight, p. 152.
117    func divWW_g(u1, u0, v Word) (q, r Word) {
118        if u1 >= v {
119            return 1<<_W - 1, 1<<_W - 1
120        }
121
122        s := leadingZeros(v)
123        v <<= s
124
125        vn1 := v >> _W2
126        vn0 := v & _M2
127        un32 := u1<<s | u0>>(_W-s)
128        un10 := u0 << s
129        un1 := un10 >> _W2
130        un0 := un10 & _M2
131        q1 := un32 / vn1
132        rhat := un32 - q1*vn1
133
134    again1:
135        if q1 >= _B2 || q1*vn0 > _B2*rhat+un1 {
136            q1--
137            rhat += vn1
138            if rhat < _B2 {
139                goto again1
140            }
141        }
142
143        un21 := un32*_B2 + un1 - q1*v

```

```

144         q0 := un21 / vn1
145         rhat = un21 - q0*vn1
146
147     again2:
148         if q0 >= _B2 || q0*vn0 > _B2*rhat+un0 {
149             q0--
150             rhat += vn1
151             if rhat < _B2 {
152                 goto again2
153             }
154         }
155
156         return q1*_B2 + q0, (un21*_B2 + un0 - q0*v) >> s
157     }
158
159     func addVV_g(z, x, y []Word) (c Word) {
160         for i := range z {
161             c, z[i] = addWW_g(x[i], y[i], c)
162         }
163         return
164     }
165
166     func subVV_g(z, x, y []Word) (c Word) {
167         for i := range z {
168             c, z[i] = subWW_g(x[i], y[i], c)
169         }
170         return
171     }
172
173     func addVW_g(z, x []Word, y Word) (c Word) {
174         c = y
175         for i := range z {
176             c, z[i] = addWW_g(x[i], c, 0)
177         }
178         return
179     }
180
181     func subVW_g(z, x []Word, y Word) (c Word) {
182         c = y
183         for i := range z {
184             c, z[i] = subWW_g(x[i], c, 0)
185         }
186         return
187     }
188
189     func sh1VU_g(z, x []Word, s uint) (c Word) {
190         if n := len(z); n > 0 {
191             s := _W - s
192             w1 := x[n-1]

```

```

193         c = w1 >> ŝ
194         for i := n - 1; i > 0; i-- {
195             w := w1
196             w1 = x[i-1]
197             z[i] = w<<s | w1>>ŝ
198         }
199         z[0] = w1 << s
200     }
201     return
202 }
203
204 func shrVU_g(z, x []Word, s uint) (c Word) {
205     if n := len(z); n > 0 {
206         ŝ := _W - s
207         w1 := x[0]
208         c = w1 << ŝ
209         for i := 0; i < n-1; i++ {
210             w := w1
211             w1 = x[i+1]
212             z[i] = w>>s | w1<<ŝ
213         }
214         z[n-1] = w1 >> s
215     }
216     return
217 }
218
219 func mulAddVWW_g(z, x []Word, y, r Word) (c Word) {
220     c = r
221     for i := range z {
222         c, z[i] = mulAddWWW_g(x[i], y, c)
223     }
224     return
225 }
226
227 func addMulVWW_g(z, x []Word, y Word) (c Word) {
228     for i := range z {
229         z1, z0 := mulAddWWW_g(x[i], y, z[i])
230         c, z[i] = addWW_g(z0, c, 0)
231         c += z1
232     }
233     return
234 }
235
236 func divVWW_g(z []Word, xn Word, x []Word, y Word) (r Word)
237     r = xn
238     for i := len(z) - 1; i >= 0; i-- {
239         z[i], r = divWW_g(r, x[i], y)
240     }
241     return
242 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/big/arith_decl.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package big
6
7 // implemented in arith_${GOARCH}.s
8 func mulWW(x, y Word) (z1, z0 Word)
9 func divWW(x1, x0, y Word) (q, r Word)
10 func addVV(z, x, y []Word) (c Word)
11 func subVV(z, x, y []Word) (c Word)
12 func addVW(z, x []Word, y Word) (c Word)
13 func subVW(z, x []Word, y Word) (c Word)
14 func shlVU(z, x []Word, s uint) (c Word)
15 func shrVU(z, x []Word, s uint) (c Word)
16 func mulAddVW(z, x []Word, y, r Word) (c Word)
17 func addMulVVW(z, x []Word, y Word) (c Word)
18 func divVW(z []Word, xn Word, x []Word, y Word) (r Word)
19 func bitLen(x Word) (n int)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/big/int.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file implements signed multi-precision integers.
6
7 package big
8
9 import (
10     "errors"
11     "fmt"
12     "io"
13     "math/rand"
14     "strings"
15 )
16
17 // An Int represents a signed multi-precision integer.
18 // The zero value for an Int represents the value 0.
19 type Int struct {
20     neg bool // sign
21     abs nat // absolute value of the integer
22 }
23
24 var intOne = &Int{false, natOne}
25
26 // Sign returns:
27 //
28 //     -1 if x < 0
29 //     0 if x == 0
30 //     +1 if x > 0
31 //
32 func (x *Int) Sign() int {
33     if len(x.abs) == 0 {
34         return 0
35     }
36     if x.neg {
37         return -1
38     }
39     return 1
40 }
41
42 // SetInt64 sets z to x and returns z.
43 func (z *Int) SetInt64(x int64) *Int {
44     neg := false
```

```

45         if x < 0 {
46             neg = true
47             x = -x
48         }
49         z.abs = z.abs.setUint64(uint64(x))
50         z.neg = neg
51         return z
52     }
53
54     // NewInt allocates and returns a new Int set to x.
55     func NewInt(x int64) *Int {
56         return new(Int).SetInt64(x)
57     }
58
59     // Set sets z to x and returns z.
60     func (z *Int) Set(x *Int) *Int {
61         if z != x {
62             z.abs = z.abs.set(x.abs)
63             z.neg = x.neg
64         }
65         return z
66     }
67
68     // Bits provides raw (unchecked but fast) access to x by ret
69     // absolute value as a little-endian word slice. The result
70     // the same underlying array.
71     // Bits is intended to support implementation of missing low
72     // functionality outside this package; it should be avoided
73     func (x *Int) Bits() []Word {
74         return x.abs
75     }
76
77     // SetBits provides raw (unchecked but fast) access to z by
78     // value to abs, interpreted as a little-endian word slice,
79     // z. The result and abs share the same underlying array.
80     // SetBits is intended to support implementation of missing
81     // functionality outside this package; it should be avoided
82     func (z *Int) SetBits(abs []Word) *Int {
83         z.abs = nat(abs).norm()
84         z.neg = false
85         return z
86     }
87
88     // Abs sets z to |x| (the absolute value of x) and returns z
89     func (z *Int) Abs(x *Int) *Int {
90         z.Set(x)
91         z.neg = false
92         return z
93     }
94

```

```

95 // Neg sets z to -x and returns z.
96 func (z *Int) Neg(x *Int) *Int {
97     z.Set(x)
98     z.neg = len(z.abs) > 0 && !z.neg // 0 has no sign
99     return z
100 }
101
102 // Add sets z to the sum x+y and returns z.
103 func (z *Int) Add(x, y *Int) *Int {
104     neg := x.neg
105     if x.neg == y.neg {
106         // x + y == x + y
107         // (-x) + (-y) == -(x + y)
108         z.abs = z.abs.add(x.abs, y.abs)
109     } else {
110         // x + (-y) == x - y == -(y - x)
111         // (-x) + y == y - x == -(x - y)
112         if x.abs.cmp(y.abs) >= 0 {
113             z.abs = z.abs.sub(x.abs, y.abs)
114         } else {
115             neg = !neg
116             z.abs = z.abs.sub(y.abs, x.abs)
117         }
118     }
119     z.neg = len(z.abs) > 0 && neg // 0 has no sign
120     return z
121 }
122
123 // Sub sets z to the difference x-y and returns z.
124 func (z *Int) Sub(x, y *Int) *Int {
125     neg := x.neg
126     if x.neg != y.neg {
127         // x - (-y) == x + y
128         // (-x) - y == -(x + y)
129         z.abs = z.abs.add(x.abs, y.abs)
130     } else {
131         // x - y == x - y == -(y - x)
132         // (-x) - (-y) == y - x == -(x - y)
133         if x.abs.cmp(y.abs) >= 0 {
134             z.abs = z.abs.sub(x.abs, y.abs)
135         } else {
136             neg = !neg
137             z.abs = z.abs.sub(y.abs, x.abs)
138         }
139     }
140     z.neg = len(z.abs) > 0 && neg // 0 has no sign
141     return z
142 }
143

```

```

144 // Mul sets z to the product x*y and returns z.
145 func (z *Int) Mul(x, y *Int) *Int {
146     // x * y == x * y
147     // x * (-y) == -(x * y)
148     // (-x) * y == -(x * y)
149     // (-x) * (-y) == x * y
150     z.abs = z.abs.mul(x.abs, y.abs)
151     z.neg = len(z.abs) > 0 && x.neg != y.neg // 0 has no
152     return z
153 }
154
155 // MulRange sets z to the product of all integers
156 // in the range [a, b] inclusively and returns z.
157 // If a > b (empty range), the result is 1.
158 func (z *Int) MulRange(a, b int64) *Int {
159     switch {
160     case a > b:
161         return z.SetInt64(1) // empty range
162     case a <= 0 && b >= 0:
163         return z.SetInt64(0) // range includes 0
164     }
165     // a <= b && (b < 0 || a > 0)
166
167     neg := false
168     if a < 0 {
169         neg = (b-a)&1 == 0
170         a, b = -b, -a
171     }
172
173     z.abs = z.abs.mulRange(uint64(a), uint64(b))
174     z.neg = neg
175     return z
176 }
177
178 // Binomial sets z to the binomial coefficient of (n, k) and
179 func (z *Int) Binomial(n, k int64) *Int {
180     var a, b Int
181     a.MulRange(n-k+1, n)
182     b.MulRange(1, k)
183     return z.Quo(&a, &b)
184 }
185
186 // Quo sets z to the quotient x/y for y != 0 and returns z.
187 // If y == 0, a division-by-zero run-time panic occurs.
188 // Quo implements truncated division (like Go); see QuoRem f
189 func (z *Int) Quo(x, y *Int) *Int {
190     z.abs, _ = z.abs.div(nil, x.abs, y.abs)
191     z.neg = len(z.abs) > 0 && x.neg != y.neg // 0 has no
192     return z

```

```

193 }
194
195 // Rem sets z to the remainder x%y for y != 0 and returns z.
196 // If y == 0, a division-by-zero run-time panic occurs.
197 // Rem implements truncated modulus (like Go); see QuoRem fo
198 func (z *Int) Rem(x, y *Int) *Int {
199     _, z.abs = nat(nil).div(z.abs, x.abs, y.abs)
200     z.neg = len(z.abs) > 0 && x.neg // 0 has no sign
201     return z
202 }
203
204 // QuoRem sets z to the quotient x/y and r to the remainder
205 // and returns the pair (z, r) for y != 0.
206 // If y == 0, a division-by-zero run-time panic occurs.
207 //
208 // QuoRem implements T-division and modulus (like Go):
209 //
210 //     q = x/y      with the result truncated to zero
211 //     r = x - y*q
212 //
213 // (See Daan Leijen, ``Division and Modulus for Computer Sci
214 // See DivMod for Euclidean division and modulus (unlike Go)
215 //
216 func (z *Int) QuoRem(x, y, r *Int) (*Int, *Int) {
217     z.abs, r.abs = z.abs.div(r.abs, x.abs, y.abs)
218     z.neg, r.neg = len(z.abs) > 0 && x.neg != y.neg, len
219     return z, r
220 }
221
222 // Div sets z to the quotient x/y for y != 0 and returns z.
223 // If y == 0, a division-by-zero run-time panic occurs.
224 // Div implements Euclidean division (unlike Go); see DivMod
225 func (z *Int) Div(x, y *Int) *Int {
226     y_neg := y.neg // z may be an alias for y
227     var r Int
228     z.QuoRem(x, y, &r)
229     if r.neg {
230         if y_neg {
231             z.Add(z, intOne)
232         } else {
233             z.Sub(z, intOne)
234         }
235     }
236     return z
237 }
238
239 // Mod sets z to the modulus x%y for y != 0 and returns z.
240 // If y == 0, a division-by-zero run-time panic occurs.
241 // Mod implements Euclidean modulus (unlike Go); see DivMod
242 func (z *Int) Mod(x, y *Int) *Int {

```

```

243     y0 := y // save y
244     if z == y || alias(z.abs, y.abs) {
245         y0 = new(Int).Set(y)
246     }
247     var q Int
248     q.QuoRem(x, y, z)
249     if z.neg {
250         if y0.neg {
251             z.Sub(z, y0)
252         } else {
253             z.Add(z, y0)
254         }
255     }
256     return z
257 }
258
259 // DivMod sets z to the quotient x div y and m to the modulus
260 // and returns the pair (z, m) for y != 0.
261 // If y == 0, a division-by-zero run-time panic occurs.
262 //
263 // DivMod implements Euclidean division and modulus (unlike
264 //
265 //     q = x div y   such that
266 //     m = x - y*q   with 0 <= m < |q|
267 //
268 // (See Raymond T. Boute, ``The Euclidean definition of the
269 // div and mod''. ACM Transactions on Programming Languages
270 // Systems (TOPLAS), 14(2):127-144, New York, NY, USA, 4/199
271 // ACM press.)
272 // See QuoRem for T-division and modulus (like Go).
273 //
274 func (z *Int) DivMod(x, y, m *Int) (*Int, *Int) {
275     y0 := y // save y
276     if z == y || alias(z.abs, y.abs) {
277         y0 = new(Int).Set(y)
278     }
279     z.QuoRem(x, y, m)
280     if m.neg {
281         if y0.neg {
282             z.Add(z, intOne)
283             m.Sub(m, y0)
284         } else {
285             z.Sub(z, intOne)
286             m.Add(m, y0)
287         }
288     }
289     return z, m
290 }
291

```

```

292 // Cmp compares x and y and returns:
293 //
294 //   -1 if x < y
295 //    0 if x == y
296 //   +1 if x > y
297 //
298 func (x *Int) Cmp(y *Int) (r int) {
299     // x cmp y == x cmp y
300     // x cmp (-y) == x
301     // (-x) cmp y == y
302     // (-x) cmp (-y) == -(x cmp y)
303     switch {
304     case x.neg == y.neg:
305         r = x.abs.cmp(y.abs)
306         if x.neg {
307             r = -r
308         }
309     case x.neg:
310         r = -1
311     default:
312         r = 1
313     }
314     return
315 }
316
317 func (x *Int) String() string {
318     switch {
319     case x == nil:
320         return "<nil>"
321     case x.neg:
322         return "-" + x.abs.decimalString()
323     }
324     return x.abs.decimalString()
325 }
326
327 func charset(ch rune) string {
328     switch ch {
329     case 'b':
330         return lowercaseDigits[0:2]
331     case 'o':
332         return lowercaseDigits[0:8]
333     case 'd', 's', 'v':
334         return lowercaseDigits[0:10]
335     case 'x':
336         return lowercaseDigits[0:16]
337     case 'X':
338         return uppercaseDigits[0:16]
339     }
340     return "" // unknown format

```

```

341 }
342
343 // write count copies of text to s
344 func writeMultiple(s fmt.State, text string, count int) {
345     if len(text) > 0 {
346         b := []byte(text)
347         for ; count > 0; count-- {
348             s.Write(b)
349         }
350     }
351 }
352
353 // Format is a support routine for fmt.Formatter. It accepts
354 // the formats 'b' (binary), 'o' (octal), 'd' (decimal), 'x'
355 // (lowercase hexadecimal), and 'X' (uppercase hexadecimal).
356 // Also supported are the full suite of package fmt's format
357 // verbs for integral types, including '+', '-', and ' '
358 // for sign control, '#' for leading zero in octal and for
359 // hexadecimal, a leading "0x" or "0X" for "%#x" and "%#X"
360 // respectively, specification of minimum digits precision,
361 // output field width, space or zero padding, and left or
362 // right justification.
363 //
364 func (x *Int) Format(s fmt.State, ch rune) {
365     cs := charset(ch)
366
367     // special cases
368     switch {
369     case cs == "":
370         // unknown format
371         fmt.Fprintf(s, "%%!c(big.Int=%s)", ch, x.St
372         return
373     case x == nil:
374         fmt.Fprint(s, "<nil>")
375         return
376     }
377
378     // determine sign character
379     sign := ""
380     switch {
381     case x.neg:
382         sign = "-"
383     case s.Flag('+'): // supersedes ' ' when both specif
384         sign = "+"
385     case s.Flag(' '):
386         sign = " "
387     }
388
389     // determine prefix characters for indicating output
390     prefix := ""

```

```

391     if s.Flag('#') {
392         switch ch {
393             case 'o': // octal
394                 prefix = "0"
395             case 'x': // hexadecimal
396                 prefix = "0x"
397             case 'X':
398                 prefix = "0X"
399         }
400     }
401
402     // determine digits with base set by len(cs) and dig
403     digits := x.abs.string(cs)
404
405     // number of characters for the three classes of num
406     var left int // space characters to left of digits
407     var zeroes int // zero characters (actually cs[0]) a
408     var right int // space characters to right of digit
409
410     // determine number padding from precision: the leas
411     precision, precisionSet := s.Precision()
412     if precisionSet {
413         switch {
414             case len(digits) < precision:
415                 zeroes = precision - len(digits) //
416             case digits == "0" && precision == 0:
417                 return // print nothing if zero valu
418         }
419     }
420
421     // determine field pad from width: the least number
422     length := len(sign) + len(prefix) + zeroes + len(dig
423     if width, widthSet := s.Width(); widthSet && length
424         switch d := width - length; {
425             case s.Flag('-'):
426                 // pad on the right with spaces; sup
427                 right = d
428             case s.Flag('0') && !precisionSet:
429                 // pad with zeroes unless precision
430                 zeroes = d
431             default:
432                 // pad on the left with spaces
433                 left = d
434         }
435     }
436
437     // print number as [left pad][sign][prefix][zero pad
438     writeMultiple(s, " ", left)
439     writeMultiple(s, sign, 1)

```

```

440         writeMultiple(s, prefix, 1)
441         writeMultiple(s, "0", zeroes)
442         writeMultiple(s, digits, 1)
443         writeMultiple(s, " ", right)
444     }
445
446     // scan sets z to the integer value corresponding to the lon
447     // read from r representing a signed integer number in a giv
448     // It returns z, the actual conversion base used, and an err
449     // error case, the value of z is undefined but the returned
450     // syntax follows the syntax of integer literals in Go.
451     //
452     // The base argument must be 0 or a value from 2 through Max
453     // is 0, the string prefix determines the actual conversion
454     // ``0x'' or ``0X'' selects base 16; the ``0'' prefix select
455     // ``0b'' or ``0B'' prefix selects base 2. Otherwise the sel
456     //
457     func (z *Int) scan(r io.RuneScanner, base int) (*Int, int, e
458         // determine sign
459         ch, _, err := r.ReadRune()
460         if err != nil {
461             return nil, 0, err
462         }
463         neg := false
464         switch ch {
465         case '-':
466             neg = true
467         case '+': // nothing to do
468         default:
469             r.UnreadRune()
470         }
471
472         // determine mantissa
473         z.abs, base, err = z.abs.scan(r, base)
474         if err != nil {
475             return nil, base, err
476         }
477         z.neg = len(z.abs) > 0 && neg // 0 has no sign
478
479         return z, base, nil
480     }
481
482     // Scan is a support routine for fmt.Scanner; it sets z to t
483     // the scanned number. It accepts the formats 'b' (binary),
484     // 'd' (decimal), 'x' (lowercase hexadecimal), and 'X' (uppe
485     func (z *Int) Scan(s fmt.ScanState, ch rune) error {
486         s.SkipSpace() // skip leading space characters
487         base := 0
488         switch ch {

```

```

489         case 'b':
490             base = 2
491         case 'o':
492             base = 8
493         case 'd':
494             base = 10
495         case 'x', 'X':
496             base = 16
497         case 's', 'v':
498             // let scan determine the base
499         default:
500             return errors.New("Int.Scan: invalid verb")
501     }
502     _, _, err := z.scan(s, base)
503     return err
504 }
505
506 // Int64 returns the int64 representation of x.
507 // If x cannot be represented in an int64, the result is und
508 func (x *Int) Int64() int64 {
509     if len(x.abs) == 0 {
510         return 0
511     }
512     v := int64(x.abs[0])
513     if _w == 32 && len(x.abs) > 1 {
514         v |= int64(x.abs[1]) << 32
515     }
516     if x.neg {
517         v = -v
518     }
519     return v
520 }
521
522 // SetString sets z to the value of s, interpreted in the gi
523 // and returns z and a boolean indicating success. If SetStr
524 // the value of z is undefined but the returned value is nil
525 //
526 // The base argument must be 0 or a value from 2 through Max
527 // is 0, the string prefix determines the actual conversion
528 // ``0x'' or ``0X'' selects base 16; the ``0'' prefix select
529 // ``0b'' or ``0B'' prefix selects base 2. Otherwise the sel
530 //
531 func (z *Int) SetString(s string, base int) (*Int, bool) {
532     r := strings.NewReader(s)
533     _, _, err := z.scan(r, base)
534     if err != nil {
535         return nil, false
536     }
537     _, _, err = r.ReadRune()
538     if err != io.EOF {

```

```

539         return nil, false
540     }
541     return z, true // err == io.EOF => scan consumed all
542 }
543
544 // SetBytes interprets buf as the bytes of a big-endian unsi
545 // integer, sets z to that value, and returns z.
546 func (z *Int) SetBytes(buf []byte) *Int {
547     z.abs = z.abs.setBytes(buf)
548     z.neg = false
549     return z
550 }
551
552 // Bytes returns the absolute value of z as a big-endian byt
553 func (x *Int) Bytes() []byte {
554     buf := make([]byte, len(x.abs)*_S)
555     return buf[x.abs.bytes(buf):]
556 }
557
558 // BitLen returns the length of the absolute value of z in b
559 // The bit length of 0 is 0.
560 func (x *Int) BitLen() int {
561     return x.abs.bitLen()
562 }
563
564 // Exp sets z = x**y mod m and returns z. If m is nil, z = x
565 // See Knuth, volume 2, section 4.6.3.
566 func (z *Int) Exp(x, y, m *Int) *Int {
567     if y.neg || len(y.abs) == 0 {
568         neg := x.neg
569         z.SetInt64(1)
570         z.neg = neg
571         return z
572     }
573
574     var mWords nat
575     if m != nil {
576         mWords = m.abs
577     }
578
579     z.abs = z.abs.expNN(x.abs, y.abs, mWords)
580     z.neg = len(z.abs) > 0 && x.neg && y.abs[0]&1 == 1 /
581     return z
582 }
583
584 // GCD sets z to the greatest common divisor of a and b, whi
585 // positive numbers, and returns z.
586 // If x and y are not nil, GCD sets x and y such that z = a*
587 // If either a or b is not positive, GCD sets z = x = y = 0.

```

```

588 func (z *Int) GCD(x, y, a, b *Int) *Int {
589     if a.neg || b.neg {
590         z.SetInt64(0)
591         if x != nil {
592             x.SetInt64(0)
593         }
594         if y != nil {
595             y.SetInt64(0)
596         }
597         return z
598     }
599
600     A := new(Int).Set(a)
601     B := new(Int).Set(b)
602
603     X := new(Int)
604     Y := new(Int).SetInt64(1)
605
606     lastX := new(Int).SetInt64(1)
607     lastY := new(Int)
608
609     q := new(Int)
610     temp := new(Int)
611
612     for len(B.abs) > 0 {
613         r := new(Int)
614         q, r = q.QuoRem(A, B, r)
615
616         A, B = B, r
617
618         temp.Set(X)
619         X.Mul(X, q)
620         X.neg = !X.neg
621         X.Add(X, lastX)
622         lastX.Set(temp)
623
624         temp.Set(Y)
625         Y.Mul(Y, q)
626         Y.neg = !Y.neg
627         Y.Add(Y, lastY)
628         lastY.Set(temp)
629     }
630
631     if x != nil {
632         *x = *lastX
633     }
634
635     if y != nil {
636         *y = *lastY

```

```

637         }
638
639         *z = *A
640         return z
641     }
642
643     // ProbablyPrime performs n Miller-Rabin tests to check whet
644     // If it returns true, x is prime with probability 1 - 1/4^n
645     // If it returns false, x is not prime.
646     func (x *Int) ProbablyPrime(n int) bool {
647         return !x.neg && x.abs.probablyPrime(n)
648     }
649
650     // Rand sets z to a pseudo-random number in [0, n) and retur
651     func (z *Int) Rand(rnd *rand.Rand, n *Int) *Int {
652         z.neg = false
653         if n.neg == true || len(n.abs) == 0 {
654             z.abs = nil
655             return z
656         }
657         z.abs = z.abs.random(rnd, n.abs, n.abs.bitLen())
658         return z
659     }
660
661     // ModInverse sets z to the multiplicative inverse of g in t
662     // p is a prime) and returns z.
663     func (z *Int) ModInverse(g, p *Int) *Int {
664         var d Int
665         d.GCD(z, nil, g, p)
666         // x and y are such that g*x + p*y = d. Since p is p
667         // that modulo p results in g*x = 1, therefore x is
668         if z.neg {
669             z.Add(z, p)
670         }
671         return z
672     }
673
674     // Lsh sets z = x << n and returns z.
675     func (z *Int) Lsh(x *Int, n uint) *Int {
676         z.abs = z.abs.shl(x.abs, n)
677         z.neg = x.neg
678         return z
679     }
680
681     // Rsh sets z = x >> n and returns z.
682     func (z *Int) Rsh(x *Int, n uint) *Int {
683         if x.neg {
684             // (-x) >> s == ^(x-1) >> s == ^((x-1) >> s)
685             t := z.abs.sub(x.abs, natOne) // no underflo
686             t = t.shr(t, n)

```

```

687             z.abs = t.add(t, natOne)
688             z.neg = true // z cannot be zero if x is neg
689             return z
690         }
691
692         z.abs = z.abs.shr(x.abs, n)
693         z.neg = false
694         return z
695     }
696
697     // Bit returns the value of the i'th bit of x. That is, it
698     // returns (x>>i)&1. The bit index i must be >= 0.
699     func (x *Int) Bit(i int) uint {
700         if i < 0 {
701             panic("negative bit index")
702         }
703         if x.neg {
704             t := nat(nil).sub(x.abs, natOne)
705             return t.bit(uint(i)) ^ 1
706         }
707
708         return x.abs.bit(uint(i))
709     }
710
711     // SetBit sets z to x, with x's i'th bit set to b (0 or 1).
712     // That is, if bit is 1 SetBit sets z = x | (1 << i);
713     // if bit is 0 it sets z = x &^ (1 << i). If bit is not 0 or
714     // SetBit will panic.
715     func (z *Int) SetBit(x *Int, i int, b uint) *Int {
716         if i < 0 {
717             panic("negative bit index")
718         }
719         if x.neg {
720             t := z.abs.sub(x.abs, natOne)
721             t = t.setBit(t, uint(i), b^1)
722             z.abs = t.add(t, natOne)
723             z.neg = len(z.abs) > 0
724         }
725
726         z.abs = z.abs.setBit(x.abs, uint(i), b)
727         z.neg = false
728         return z
729     }
730
731     // And sets z = x & y and returns z.
732     func (z *Int) And(x, y *Int) *Int {
733         if x.neg == y.neg {
734             if x.neg {
735                 // (-x) & (-y) == ^(x-1) & ^(y-1) ==

```

```

736             x1 := nat(nil).sub(x.abs, natOne)
737             y1 := nat(nil).sub(y.abs, natOne)
738             z.abs = z.abs.add(z.abs.or(x1, y1),
739             z.neg = true // z cannot be zero if
740             return z
741         }
742
743         // x & y == x & y
744         z.abs = z.abs.and(x.abs, y.abs)
745         z.neg = false
746         return z
747     }
748
749     // x.neg != y.neg
750     if x.neg {
751         x, y = y, x // & is symmetric
752     }
753
754     // x & (-y) == x & ^ (y-1) == x & ^ (y-1)
755     y1 := nat(nil).sub(y.abs, natOne)
756     z.abs = z.abs.andNot(x.abs, y1)
757     z.neg = false
758     return z
759 }
760
761 // AndNot sets z = x & ^ y and returns z.
762 func (z *Int) AndNot(x, y *Int) *Int {
763     if x.neg == y.neg {
764         if x.neg {
765             // (-x) & ^ (-y) == ^ (x-1) & ^ (y-1)
766             x1 := nat(nil).sub(x.abs, natOne)
767             y1 := nat(nil).sub(y.abs, natOne)
768             z.abs = z.abs.andNot(y1, x1)
769             z.neg = false
770             return z
771         }
772
773         // x & ^ y == x & ^ y
774         z.abs = z.abs.andNot(x.abs, y.abs)
775         z.neg = false
776         return z
777     }
778
779     if x.neg {
780         // (-x) & ^ y == ^ (x-1) & ^ y == ^ (x-1) & ^ y =
781         x1 := nat(nil).sub(x.abs, natOne)
782         z.abs = z.abs.add(z.abs.or(x1, y.abs), natOne)
783         z.neg = true // z cannot be zero if x is neg
784         return z

```

```

785     }
786
787     //  $x \&^{\neg y} == x \&^{\wedge(y-1)} == x \& (y-1)$ 
788     y1 := nat(nil).add(y.abs, natOne)
789     z.abs = z.abs.and(x.abs, y1)
790     z.neg = false
791     return z
792 }
793
794 // Or sets  $z = x \mid y$  and returns z.
795 func (z *Int) Or(x, y *Int) *Int {
796     if x.neg == y.neg {
797         if x.neg {
798             //  $(\neg x) \mid (\neg y) == \wedge(x-1) \mid \wedge(y-1) ==$ 
799             x1 := nat(nil).sub(x.abs, natOne)
800             y1 := nat(nil).sub(y.abs, natOne)
801             z.abs = z.abs.add(z.abs.and(x1, y1),
802             z.neg = true // z cannot be zero if
803             return z
804         }
805
806         //  $x \mid y == x \mid y$ 
807         z.abs = z.abs.or(x.abs, y.abs)
808         z.neg = false
809         return z
810     }
811
812     //  $x.neg \neq y.neg$ 
813     if x.neg {
814         x, y = y, x //  $\mid$  is symmetric
815     }
816
817     //  $x \mid (\neg y) == x \mid \wedge(y-1) == \wedge((y-1) \&^{\neg x}) == \neg(\wedge(y$ 
818     y1 := nat(nil).sub(y.abs, natOne)
819     z.abs = z.abs.add(z.abs.andNot(y1, x.abs), natOne)
820     z.neg = true // z cannot be zero if one of x or y is
821     return z
822 }
823
824 // Xor sets  $z = x \wedge y$  and returns z.
825 func (z *Int) Xor(x, y *Int) *Int {
826     if x.neg == y.neg {
827         if x.neg {
828             //  $(\neg x) \wedge (\neg y) == \wedge(x-1) \wedge \wedge(y-1) ==$ 
829             x1 := nat(nil).sub(x.abs, natOne)
830             y1 := nat(nil).sub(y.abs, natOne)
831             z.abs = z.abs.xor(x1, y1)
832             z.neg = false
833             return z
834         }

```

```

835
836         // x ^ y == x ^ y
837         z.abs = z.abs.xor(x.abs, y.abs)
838         z.neg = false
839         return z
840     }
841
842     // x.neg != y.neg
843     if x.neg {
844         x, y = y, x // ^ is symmetric
845     }
846
847     // x ^ (-y) == x ^ ^ (y-1) == ^ (x ^ (y-1)) == -((x ^
848     y1 := nat(nil).sub(y.abs, natOne)
849     z.abs = z.abs.add(z.abs.xor(x.abs, y1), natOne)
850     z.neg = true // z cannot be zero if only one of x or
851     return z
852 }
853
854 // Not sets z = ^x and returns z.
855 func (z *Int) Not(x *Int) *Int {
856     if x.neg {
857         // ^(-x) == ^(^ (x-1)) == x-1
858         z.abs = z.abs.sub(x.abs, natOne)
859         z.neg = false
860         return z
861     }
862
863     // ^x == -x-1 == -(x+1)
864     z.abs = z.abs.add(x.abs, natOne)
865     z.neg = true // z cannot be zero if x is positive
866     return z
867 }
868
869 // Gob codec version. Permits backward-compatible changes to
870 const intGobVersion byte = 1
871
872 // GobEncode implements the gob.GobEncoder interface.
873 func (x *Int) GobEncode() ([]byte, error) {
874     buf := make([]byte, 1+len(x.abs)*_S) // extra byte f
875     i := x.abs.bytes(buf) - 1           // i >= 0
876     b := intGobVersion << 1             // make space f
877     if x.neg {
878         b |= 1
879     }
880     buf[i] = b
881     return buf[i:], nil
882 }
883

```

```
884 // GobDecode implements the gob.GobDecoder interface.
885 func (z *Int) GobDecode(buf []byte) error {
886     if len(buf) == 0 {
887         return errors.New("Int.GobDecode: no data")
888     }
889     b := buf[0]
890     if b>>1 != intGobVersion {
891         return errors.New(fmt.Sprintf("Int.GobDecode
892     })
893     z.neg = b&1 != 0
894     z.abs = z.abs.setBytes(buf[1:])
895     return nil
896 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/big/nat.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package big implements multi-precision arithmetic (big nu
6 // The following numeric types are supported:
7 //
8 //     - Int    signed integers
9 //     - Rat    rational numbers
10 //
11 // Methods are typically of the form:
12 //
13 //     func (z *Int) Op(x, y *Int) *Int    (similar for
14 //
15 // and implement operations  $z = x \text{ Op } y$  with the result as re
16 // is one of the operands it may be overwritten (and its mem
17 // To enable chaining of operations, the result is also retu
18 // returning a result other than *Int or *Rat take one of th
19 // the receiver.
20 //
21 package big
22
23 // This file contains operations on unsigned multi-precision
24 // These are the building blocks for the operations on signe
25 // and rationals.
26
27 import (
28     "errors"
29     "io"
30     "math"
31     "math/rand"
32     "sync"
33 )
34
35 // An unsigned integer x of the form
36 //
37 //      $x = x[n-1] \cdot \_B^{(n-1)} + x[n-2] \cdot \_B^{(n-2)} + \dots + x[1] \cdot \_B +$ 
38 //
39 // with  $0 \leq x[i] < \_B$  and  $0 \leq i < n$  is stored in a slice o
40 // with the digits  $x[i]$  as the slice elements.
41 //
42 // A number is normalized if the slice contains no leading 0
43 // During arithmetic operations, denormalized values may occ
44 // always normalized before returning the final result. The
```

```

45 // representation of 0 is the empty or nil slice (length = 0
46 //
47 type nat []Word
48
49 var (
50     natOne = nat{1}
51     natTwo = nat{2}
52     natTen = nat{10}
53 )
54
55 func (z nat) clear() {
56     for i := range z {
57         z[i] = 0
58     }
59 }
60
61 func (z nat) norm() nat {
62     i := len(z)
63     for i > 0 && z[i-1] == 0 {
64         i--
65     }
66     return z[0:i]
67 }
68
69 func (z nat) make(n int) nat {
70     if n <= cap(z) {
71         return z[0:n] // reuse z
72     }
73     // Choosing a good value for e has significant perfo
74     // because it increases the chance that a value can
75     const e = 4 // extra capacity
76     return make(nat, n, n+e)
77 }
78
79 func (z nat) setWord(x Word) nat {
80     if x == 0 {
81         return z.make(0)
82     }
83     z = z.make(1)
84     z[0] = x
85     return z
86 }
87
88 func (z nat) setUint64(x uint64) nat {
89     // single-digit values
90     if w := Word(x); uint64(w) == x {
91         return z.setWord(w)
92     }
93
94     // compute number of words n required to represent x

```

```

95         n := 0
96         for t := x; t > 0; t >>= _W {
97             n++
98         }
99
100        // split x into n words
101        z = z.make(n)
102        for i := range z {
103            z[i] = Word(x & _M)
104            x >>= _W
105        }
106
107        return z
108    }
109
110    func (z nat) set(x nat) nat {
111        z = z.make(len(x))
112        copy(z, x)
113        return z
114    }
115
116    func (z nat) add(x, y nat) nat {
117        m := len(x)
118        n := len(y)
119
120        switch {
121        case m < n:
122            return z.add(y, x)
123        case m == 0:
124            // n == 0 because m >= n; result is 0
125            return z.make(0)
126        case n == 0:
127            // result is x
128            return z.set(x)
129        }
130        // m > 0
131
132        z = z.make(m + 1)
133        c := addVV(z[0:n], x, y)
134        if m > n {
135            c = addVW(z[n:m], x[n:], c)
136        }
137        z[m] = c
138
139        return z.norm()
140    }
141
142    func (z nat) sub(x, y nat) nat {
143        m := len(x)

```

```

144         n := len(y)
145
146         switch {
147         case m < n:
148             panic("underflow")
149         case m == 0:
150             // n == 0 because m >= n; result is 0
151             return z.make(0)
152         case n == 0:
153             // result is x
154             return z.set(x)
155         }
156         // m > 0
157
158         z = z.make(m)
159         c := subVV(z[0:n], x, y)
160         if m > n {
161             c = subVW(z[n:], x[n:], c)
162         }
163         if c != 0 {
164             panic("underflow")
165         }
166
167         return z.norm()
168     }
169
170     func (x nat) cmp(y nat) (r int) {
171         m := len(x)
172         n := len(y)
173         if m != n || m == 0 {
174             switch {
175             case m < n:
176                 r = -1
177             case m > n:
178                 r = 1
179             }
180             return
181         }
182
183         i := m - 1
184         for i > 0 && x[i] == y[i] {
185             i--
186         }
187
188         switch {
189         case x[i] < y[i]:
190             r = -1
191         case x[i] > y[i]:
192             r = 1

```

```

193     }
194     return
195 }
196
197 func (z nat) mulAddWW(x nat, y, r Word) nat {
198     m := len(x)
199     if m == 0 || y == 0 {
200         return z.setWord(r) // result is r
201     }
202     // m > 0
203
204     z = z.make(m + 1)
205     z[m] = mulAddVWW(z[0:m], x, y, r)
206
207     return z.norm()
208 }
209
210 // basicMul multiplies x and y and leaves the result in z.
211 // The (non-normalized) result is placed in z[0 : len(x) + 1
212 func basicMul(z, x, y nat) {
213     z[0 : len(x)+len(y)].clear() // initialize z
214     for i, d := range y {
215         if d != 0 {
216             z[len(x)+i] = addMulVWW(z[i:i+len(x)
217         }
218     }
219 }
220
221 // Fast version of z[0:n+n>>1].add(z[0:n+n>>1], x[0:n]) w/o
222 // Factored out for readability - do not use outside karatsu
223 func karatsubaAdd(z, x nat, n int) {
224     if c := addVV(z[0:n], z, x); c != 0 {
225         addVW(z[n:n+n>>1], z[n:], c)
226     }
227 }
228
229 // Like karatsubaAdd, but does subtract.
230 func karatsubaSub(z, x nat, n int) {
231     if c := subVV(z[0:n], z, x); c != 0 {
232         subVW(z[n:n+n>>1], z[n:], c)
233     }
234 }
235
236 // Operands that are shorter than karatsubaThreshold are mul
237 // "grade school" multiplication; for longer operands the Ka
238 // is used.
239 var karatsubaThreshold int = 32 // computed by calibrate.go
240
241 // karatsuba multiplies x and y and leaves the result in z.
242 // Both x and y must have the same length n and n must be a

```

```

243 // power of 2. The result vector z must have len(z) >= 6*n.
244 // The (non-normalized) result is placed in z[0 : 2*n].
245 func karatsuba(z, x, y nat) {
246     n := len(y)
247
248     // Switch to basic multiplication if numbers are odd
249     // (n is always even if karatsubaThreshold is even,
250     // conservative)
251     if n&1 != 0 || n < karatsubaThreshold || n < 2 {
252         basicMul(z, x, y)
253         return
254     }
255     // n&1 == 0 && n >= karatsubaThreshold && n >= 2
256
257     // Karatsuba multiplication is based on the observat
258     // for two numbers x and y with:
259     //
260     //   x = x1*b + x0
261     //   y = y1*b + y0
262     //
263     // the product x*y can be obtained with 3 products z
264     // instead of 4:
265     //
266     //   x*y = x1*y1*b*b + (x1*y0 + x0*y1)*b + x0*y0
267     //         =      z2*b*b +           z1*b +      z0
268     //
269     // with:
270     //
271     //   xd = x1 - x0
272     //   yd = y0 - y1
273     //
274     //   z1 =      xd*yd                + z1 + z0
275     //         = (x1-x0)*(y0 - y1)      + z1 + z0
276     //         = x1*y0 - x1*y1 - x0*y0 + x0*y1 + z1 + z0
277     //         = x1*y0 -      z1 -      z0 + x0*y1 + z1 + z0
278     //         = x1*y0                + x0*y1
279
280     // split x, y into "digits"
281     n2 := n >> 1 // n2 >= 1
282     x1, x0 := x[n2:], x[0:n2] // x = x1*b + y0
283     y1, y0 := y[n2:], y[0:n2] // y = y1*b + y0
284
285     // z is used for the result and temporary storage:
286     //
287     //   6*n   5*n   4*n   3*n   2*n   1*n
288     // z = [z2 copy|z0 copy| xd*yd | yd:xd | x1*y1 | x0*
289     //
290     // For each recursive call of karatsuba, an unused s
291     // z is passed in that has (at least) half the lengt

```

```

292 // caller's z.
293
294 // compute z0 and z2 with the result "in place" in z
295 karatsuba(z, x0, y0) // z0 = x0*y0
296 karatsuba(z[n:], x1, y1) // z2 = x1*y1
297
298 // compute xd (or the negative value if underflow oc
299 s := 1 // sign of product xd*yd
300 xd := z[2*n : 2*n+n2]
301 if subVV(xd, x1, x0) != 0 { // x1-x0
302     s = -s
303     subVV(xd, x0, x1) // x0-x1
304 }
305
306 // compute yd (or the negative value if underflow oc
307 yd := z[2*n+n2 : 3*n]
308 if subVV(yd, y0, y1) != 0 { // y0-y1
309     s = -s
310     subVV(yd, y1, y0) // y1-y0
311 }
312
313 // p = (x1-x0)*(y0-y1) == x1*y0 - x1*y1 - x0*y0 + x0
314 // p = (x0-x1)*(y0-y1) == x0*y0 - x0*y1 - x1*y0 + x1
315 p := z[n*3:]
316 karatsuba(p, xd, yd)
317
318 // save original z2:z0
319 // (ok to use upper half of z since we're done recur
320 r := z[n*4:]
321 copy(r, z)
322
323 // add up all partial products
324 //
325 //      2*n      n      0
326 // z = [ z2 | z0 ]
327 //   + [ z0 ]
328 //   + [ z2 ]
329 //   + [ p ]
330 //
331 karatsubaAdd(z[n2:], r, n)
332 karatsubaAdd(z[n2:], r[n:], n)
333 if s > 0 {
334     karatsubaAdd(z[n2:], p, n)
335 } else {
336     karatsubaSub(z[n2:], p, n)
337 }
338 }
339
340 // alias returns true if x and y share the same base array.

```

```

341 func alias(x, y nat) bool {
342     return cap(x) > 0 && cap(y) > 0 && &x[0:cap(x)][cap(
343 }
344
345 // addAt implements z += x*(1<<(_W*i)); z must be long enoug
346 // (we don't use nat.add because we need z to stay the same
347 // slice, and we don't need to normalize z after each additi
348 func addAt(z, x nat, i int) {
349     if n := len(x); n > 0 {
350         if c := addVV(z[i:i+n], z[i:], x); c != 0 {
351             j := i + n
352             if j < len(z) {
353                 addVW(z[j:], z[j:], c)
354             }
355         }
356     }
357 }
358
359 func max(x, y int) int {
360     if x > y {
361         return x
362     }
363     return y
364 }
365
366 // karatsubaLen computes an approximation to the maximum k <
367 // k = p<<i for a number p <= karatsubaThreshold and an i >=
368 // result is the largest number that can be divided repeated
369 // becoming about the value of karatsubaThreshold.
370 func karatsubaLen(n int) int {
371     i := uint(0)
372     for n > karatsubaThreshold {
373         n >>= 1
374         i++
375     }
376     return n << i
377 }
378
379 func (z nat) mul(x, y nat) nat {
380     m := len(x)
381     n := len(y)
382
383     switch {
384     case m < n:
385         return z.mul(y, x)
386     case m == 0 || n == 0:
387         return z.make(0)
388     case n == 1:
389         return z.mulAddWW(x, y[0], 0)
390     }

```

```

391 // m >= n > 1
392
393 // determine if z can be reused
394 if alias(z, x) || alias(z, y) {
395     z = nil // z is an alias for x or y - cannot
396 }
397
398 // use basic multiplication if the numbers are small
399 if n < karatsubaThreshold || n < 2 {
400     z = z.make(m + n)
401     basicMul(z, x, y)
402     return z.norm()
403 }
404 // m >= n && n >= karatsubaThreshold && n >= 2
405
406 // determine Karatsuba length k such that
407 //
408 // x = x1*b + x0
409 // y = y1*b + y0 (and k <= len(y), which implies
410 // b = 1<<(_W*k) ("base" of digits xi, yi)
411 //
412 k := karatsubaLen(n)
413 // k <= n
414
415 // multiply x0 and y0 via Karatsuba
416 x0 := x[0:k] // x0 is not normalized
417 y0 := y[0:k] // y0 is not normalized
418 z = z.make(max(6*k, m+n)) // enough space for karats
419 karatsuba(z, x0, y0)
420 z = z[0 : m+n] // z has final length but may be inco
421
422 // If x1 and/or y1 are not 0, add missing terms to z
423 //
424 //      m+n      2*k      0
425 // z = [ ... | x0*y0 ]
426 //      + [ x1*y1 ]
427 //      + [ x1*y0 ]
428 //      + [ x0*y1 ]
429 //
430 if k < n || m != n {
431     x1 := x[k:] // x1 is normalized because x is
432     y1 := y[k:] // y1 is normalized because y is
433     var t nat
434     t = t.mul(x1, y1)
435     copy(z[2*k:], t)
436     z[2*k+len(t):].clear() // upper portion of z
437     t = t.mul(x1, y0.norm())
438     addAt(z, t, k)
439     t = t.mul(x0.norm(), y1)

```

```

440         addAt(z, t, k)
441     }
442
443     return z.norm()
444 }
445
446 // mulRange computes the product of all the unsigned integer
447 // range [a, b] inclusively. If a > b (empty range), the res
448 func (z nat) mulRange(a, b uint64) nat {
449     switch {
450     case a == 0:
451         // cut long ranges short (optimization)
452         return z.setUint64(0)
453     case a > b:
454         return z.setUint64(1)
455     case a == b:
456         return z.setUint64(a)
457     case a+1 == b:
458         return z.mul(nat(nil).setUint64(a), nat(nil))
459     }
460     m := (a + b) / 2
461     return z.mul(nat(nil).mulRange(a, m), nat(nil).mulRa
462 }
463
464 // q = (x-r)/y, with 0 <= r < y
465 func (z nat) divW(x nat, y Word) (q nat, r Word) {
466     m := len(x)
467     switch {
468     case y == 0:
469         panic("division by zero")
470     case y == 1:
471         q = z.set(x) // result is x
472         return
473     case m == 0:
474         q = z.make(0) // result is 0
475         return
476     }
477     // m > 0
478     z = z.make(m)
479     r = divWW(z, 0, x, y)
480     q = z.norm()
481     return
482 }
483
484 func (z nat) div(z2, u, v nat) (q, r nat) {
485     if len(v) == 0 {
486         panic("division by zero")
487     }
488

```

```

489         if u.cmp(v) < 0 {
490             q = z.make(0)
491             r = z2.set(u)
492             return
493         }
494
495         if len(v) == 1 {
496             var rprime Word
497             q, rprime = z.divW(u, v[0])
498             if rprime > 0 {
499                 r = z2.make(1)
500                 r[0] = rprime
501             } else {
502                 r = z2.make(0)
503             }
504             return
505         }
506
507         q, r = z.divLarge(z2, u, v)
508         return
509     }
510
511     // q = (uIn-r)/v, with 0 <= r < y
512     // Uses z as storage for q, and u as storage for r if possib
513     // See Knuth, Volume 2, section 4.3.1, Algorithm D.
514     // Preconditions:
515     //     len(v) >= 2
516     //     len(uIn) >= len(v)
517     func (z nat) divLarge(u, uIn, v nat) (q, r nat) {
518         n := len(v)
519         m := len(uIn) - n
520
521         // determine if z can be reused
522         // TODO(gri) should find a better solution - this if
523         //           is very costly (see e.g. time pidigits
524         if alias(z, uIn) || alias(z, v) {
525             z = nil // z is an alias for uIn or v - cann
526         }
527         q = z.make(m + 1)
528
529         qhatv := make(nat, n+1)
530         if alias(u, uIn) || alias(u, v) {
531             u = nil // u is an alias for uIn or v - cann
532         }
533         u = u.make(len(uIn) + 1)
534         u.clear()
535
536         // D1.
537         shift := leadingZeros(v[n-1])
538         if shift > 0 {

```

```

539         // do not modify v, it may be used by anothe
540         v1 := make(nat, n)
541         shlVU(v1, v, shift)
542         v = v1
543     }
544     u[len(uIn)] = shlVU(u[0:len(uIn)], uIn, shift)
545
546     // D2.
547     for j := m; j >= 0; j-- {
548         // D3.
549         qhat := Word(_M)
550         if u[j+n] != v[n-1] {
551             var rhat Word
552             qhat, rhat = divWW(u[j+n], u[j+n-1],
553
554                 // x1 | x2 = qv_{n-2}
555                 x1, x2 := mulWW(qhat, v[n-2])
556                 // test if qv_{n-2} > br + u_{j+n-2}
557                 for greaterThan(x1, x2, rhat, u[j+n-
558                     qhat--
559                     prevRhat := rhat
560                     rhat += v[n-1]
561                     // v[n-1] >= 0, so this test
562                     if rhat < prevRhat {
563                         break
564                     }
565                     x1, x2 = mulWW(qhat, v[n-2])
566                 }
567             }
568
569         // D4.
570         qhatv[n] = mulAddVWW(qhatv[0:n], v, qhat, 0)
571
572         c := subVV(u[j:j+len(qhatv)], u[j:], qhatv)
573         if c != 0 {
574             c := addVV(u[j:j+n], u[j:], v)
575             u[j+n] += c
576             qhat--
577         }
578
579         q[j] = qhat
580     }
581
582     q = q.norm()
583     shrVU(u, u, shift)
584     r = u.norm()
585
586     return q, r
587 }

```

```

588
589 // Length of x in bits. x must be normalized.
590 func (x nat) bitLen() int {
591     if i := len(x) - 1; i >= 0 {
592         return i*_W + bitLen(x[i])
593     }
594     return 0
595 }
596
597 // MaxBase is the largest number base accepted for string co
598 const MaxBase = 'z' - 'a' + 10 + 1 // = hexValue('z') + 1
599
600 func hexValue(ch rune) Word {
601     d := int(MaxBase + 1) // illegal base
602     switch {
603     case '0' <= ch && ch <= '9':
604         d = int(ch - '0')
605     case 'a' <= ch && ch <= 'z':
606         d = int(ch - 'a' + 10)
607     case 'A' <= ch && ch <= 'Z':
608         d = int(ch - 'A' + 10)
609     }
610     return Word(d)
611 }
612
613 // scan sets z to the natural number corresponding to the lo
614 // read from r representing an unsigned integer in a given c
615 // It returns z, the actual conversion base used, and an err
616 // error case, the value of z is undefined. The syntax follo
617 // unsigned integer literals in Go.
618 //
619 // The base argument must be 0 or a value from 2 through Max
620 // is 0, the string prefix determines the actual conversion
621 // ``0x'' or ``0X'' selects base 16; the ``0'' prefix select
622 // ``0b'' or ``0B'' prefix selects base 2. Otherwise the sel
623 //
624 func (z nat) scan(r io.RuneScanner, base int) (nat, int, err
625 // reject illegal bases
626 if base < 0 || base == 1 || MaxBase < base {
627     return z, 0, errors.New("illegal number base
628 }
629
630 // one char look-ahead
631 ch, _, err := r.ReadRune()
632 if err != nil {
633     return z, 0, err
634 }
635
636 // determine base if necessary

```

```

637     b := Word(base)
638     if base == 0 {
639         b = 10
640         if ch == '0' {
641             switch ch, _, err = r.ReadRune(); er
642             case nil:
643                 b = 8
644                 switch ch {
645                     case 'x', 'X':
646                         b = 16
647                     case 'b', 'B':
648                         b = 2
649                 }
650                 if b == 2 || b == 16 {
651                     if ch, _, err = r.Re
652                         return z, 0,
653                     }
654                 }
655             case io.EOF:
656                 return z.make(0), 10, nil
657             default:
658                 return z, 10, err
659         }
660     }
661 }
662
663 // convert string
664 // - group as many digits d as possible together int
665 // - only when bb does not fit into a word anymore,
666 z = z.make(0)
667 bb := Word(1)
668 dd := Word(0)
669 for max := _M / b; ; {
670     d := hexValue(ch)
671     if d >= b {
672         r.UnreadRune() // ch does not belong
673         break
674     }
675
676     if bb <= max {
677         bb *= b
678         dd = dd*b + d
679     } else {
680         // bb * b would overflow
681         z = z.mulAddWW(z, bb, dd)
682         bb = b
683         dd = d
684     }
685
686     if ch, _, err = r.ReadRune(); err != nil {

```

```

687             if err != io.EOF {
688                 return z, int(b), err
689             }
690             break
691         }
692     }
693
694     switch {
695     case bb > 1:
696         // there was at least one mantissa digit
697         z = z.mulAddWW(z, bb, dd)
698     case base == 0 && b == 8:
699         // there was only the octal prefix 0 (possib
700         // return base 10, not 8
701         return z, 10, nil
702     case base != 0 || b != 8:
703         // there was neither a mantissa digit nor th
704         return z, int(b), errors.New("syntax error s
705     }
706
707     return z.norm(), int(b), nil
708 }
709
710 // Character sets for string conversion.
711 const (
712     lowercaseDigits = "0123456789abcdefghijklmnopqrstuvw
713     uppercaseDigits = "0123456789ABCDEFGHIJKLMNQPQRSTUVWXYZ
714 )
715
716 // decimalString returns a decimal representation of x.
717 // It calls x.string with the charset "0123456789".
718 func (x nat) decimalString() string {
719     return x.string(lowercaseDigits[0:10])
720 }
721
722 // string converts x to a string using digits from a charset
723 // value d is represented by charset[d]. The conversion base
724 // by len(charset), which must be >= 2 and <= 256.
725 func (x nat) string(charset string) string {
726     b := Word(len(charset))
727
728     // special cases
729     switch {
730     case b < 2 || MaxBase > 256:
731         panic("illegal base")
732     case len(x) == 0:
733         return string(charset[0])
734     }
735

```

```

736 // allocate buffer for conversion
737 i := int(float64(x.bitLen())/math.Log2(float64(b)))
738 s := make([]byte, i)
739
740 // convert power of two and non power of two bases s
741 if b == b&-b {
742     // shift is base-b digit size in bits
743     shift := uint(trailingZeroBits(b)) // shift
744     mask := Word(1)<<shift - 1
745     w := x[0]
746     nbits := uint(_W) // number of unprocessed b
747
748     // convert less-significant words
749     for k := 1; k < len(x); k++ {
750         // convert full digits
751         for nbits >= shift {
752             i--
753             s[i] = charset[w&mask]
754             w >>= shift
755             nbits -= shift
756         }
757
758         // convert any partial leading digit
759         if nbits == 0 {
760             // no partial digit remainin
761             w = x[k]
762             nbits = _W
763         } else {
764             // partial digit in current
765             w |= x[k] << nbits
766             i--
767             s[i] = charset[w&mask]
768
769             // advance
770             w = x[k] >> (shift - nbits)
771             nbits = _W - (shift - nbits)
772         }
773     }
774
775     // convert digits of most-significant word (
776     for nbits >= 0 && w != 0 {
777         i--
778         s[i] = charset[w&mask]
779         w >>= shift
780         nbits -= shift
781     }
782
783 } else {
784     // determine "big base"; i.e., the largest p

```

```

785         // that is a power of base b and still fits
786         // (as in 10^19 for 19 decimal digits in a 6
787         bb := b          // big base is b**ndigits
788         ndigits := 1 // number of base b digits
789         for max := Word(_M / b); bb <= max; bb *= b
790             ndigits++ // maximize ndigits where
791         }
792
793         // construct table of successive squares of
794         // result (table != nil) <=> (len(x) > leafS
795         table := divisors(len(x), b, ndigits, bb)
796
797         // preserve x, create local copy for use by
798         q := nat(nil).set(x)
799
800         // convert q to string s in base b
801         q.convertWords(s, charset, b, ndigits, bb, t
802
803         // strip leading zeros
804         // (x != 0; thus s must contain at least one
805         // and the loop will terminate)
806         i = 0
807         for zero := charset[0]; s[i] == zero; {
808             i++
809         }
810     }
811
812     return string(s[i:])
813 }
814
815 // Convert words of q to base b digits in s. If q is large,
816 // by nat/nat division using tabulated divisors. Otherwise,
817 // repeated nat/Word divison.
818 //
819 // The iterative method processes n Words by n divW() calls,
820 // incrementally shortened q for a total of n + (n-1) + (n-2
821 // Recursive conversion divides q by its approximate square
822 // the size of q. Using the iterative method on both halves
823 // plus the expensive long div(). Asymptotically, the ratio
824 // is made better by splitting the subblocks recursively. Be
825 // split would take longer (because of the nat/nat div()) th
826 // iterative approach. This threshold is represented by leaf
827 // range 2..64 shows that values of 8 and 16 work well, with
828 // ~30x for 20000 digits. Use nat_test.go's BenchmarkLeafSiz
829 // specific hardware.
830 //
831 func (q nat) convertWords(s []byte, charset string, b Word,
832     // split larger blocks recursively
833     if table != nil {
834         // len(q) > leafSize > 0

```

```

835         var r nat
836         index := len(table) - 1
837         for len(q) > leafSize {
838             // find divisor close to sqrt(q) if
839             maxLength := q.bitLen() // ~= lc
840             minLength := maxLength >> 1 // ~= lc
841             for index > 0 && table[index-1].nbit
842                 index-- // desired
843             }
844             if table[index].nbits >= maxLength &
845                 index--
846                 if index < 0 {
847                     panic("internal inco
848                 }
849             }
850
851             // split q into the two digit number
852             q, r = q.div(r, q, table[index].bbb)
853
854             // convert subblocks and collect res
855             h := len(s) - table[index].ndigits
856             r.convertWords(s[h:], charset, b, nd
857             s = s[:h] // == q.convertWords(s, ch
858         }
859     }
860
861     // having split any large blocks now process the rem
862     i := len(s)
863     var r Word
864     if b == 10 {
865         // hard-coding for 10 here speeds this up by
866         for len(q) > 0 {
867             // extract least significant, base b
868             q, r = q.divW(q, bb)
869             for j := 0; j < ndigits && i > 0; j+
870                 i--
871                 // avoid % computation since
872                 // this appears to be faster
873                 // and smaller strings (but
874                 t := r / 10
875                 s[i] = charset[r-t<<3-t-t] /
876                 r = t
877             }
878         }
879     } else {
880         for len(q) > 0 {
881             // extract least significant, base b
882             q, r = q.divW(q, bb)
883             for j := 0; j < ndigits && i > 0; j+

```

```

884             i--
885             s[i] = charset[r%b]
886             r /= b
887         }
888     }
889 }
890
891 // prepend high-order zeroes
892 zero := charset[0]
893 for i > 0 { // while need more leading zeroes
894     i--
895     s[i] = zero
896 }
897 }
898
899 // Split blocks greater than leafSize words (or set to 0 to
900 // Benchmark and configure leafSize using: go test -bench="L
901 // 8 and 16 effective on 3.0 GHz Xeon "Clovertown" CPU (12
902 // 8 and 16 effective on 2.66 GHz Core 2 Duo "Penryn" CPU
903 var leafSize int = 8 // number of Word-size binary values tr
904
905 type divisor struct {
906     bbb      nat // divisor
907     nbits   int // bit length of divisor (discounting le
908     ndigits int // digit length of divisor in terms of c
909 }
910
911 var cacheBase10 [64]divisor // cached divisors for base 10
912 var cacheLock sync.Mutex    // protects cacheBase10
913
914 // expWW computes x**y
915 func (z nat) expWW(x, y Word) nat {
916     return z.expNN(nat(nil).setWord(x), nat(nil).setWord
917 }
918
919 // construct table of powers of bb*leafSize to use in subdiv
920 func divisors(m int, b Word, ndigits int, bb Word) []divisor
921 // only compute table when recursive conversion is e
922 if leafSize == 0 || m <= leafSize {
923     return nil
924 }
925
926 // determine k where (bb**leafSize)**(2**k) >= sqrt(
927 k := 1
928 for words := leafSize; words < m>>1 && k < len(cache
929     k++
930 }
931
932 // create new table of divisors or extend and reuse

```

```

933     var table []divisor
934     var cached bool
935     switch b {
936     case 10:
937         table = cacheBase10[0:k] // reuse old table
938         cached = true
939     default:
940         table = make([]divisor, k) // new table for
941     }
942
943     // extend table
944     if table[k-1].ndigits == 0 {
945         if cached {
946             cacheLock.Lock() // begin critical s
947         }
948
949         // add new entries as needed
950         var larger nat
951         for i := 0; i < k; i++ {
952             if table[i].ndigits == 0 {
953                 if i == 0 {
954                     table[i].bbb = nat(n
955                     table[i].ndigits = n
956                 } else {
957                     table[i].bbb = nat(n
958                     table[i].ndigits = 2
959                 }
960
961                 // optimization: exploit agg
962                 larger = nat(nil).set(table[
963                 for mulAddVwW(larger, larger
964                     table[i].bbb = table
965                     table[i].ndigits++
966                 }
967
968                 table[i].nbits = table[i].bb
969             }
970         }
971
972         if cached {
973             cacheLock.Unlock() // end critical s
974         }
975     }
976
977     return table
978 }
979
980 const deBruijn32 = 0x077CB531
981
982 var deBruijn32Lookup = []byte{

```

```

983         0, 1, 28, 2, 29, 14, 24, 3, 30, 22, 20, 15, 25, 17,
984         31, 27, 13, 23, 21, 19, 16, 7, 26, 12, 18, 6, 11, 5,
985     }
986
987     const deBruijn64 = 0x03f79d71b4ca8b09
988
989     var deBruijn64Lookup = []byte{
990         0, 1, 56, 2, 57, 49, 28, 3, 61, 58, 42, 50, 38, 29,
991         62, 47, 59, 36, 45, 43, 51, 22, 53, 39, 33, 30, 24,
992         63, 55, 48, 27, 60, 41, 37, 16, 46, 35, 44, 21, 52,
993         54, 26, 40, 15, 34, 20, 31, 10, 25, 14, 19, 9, 13, 8
994     }
995
996     // trailingZeroBits returns the number of consecutive zero b
997     // side of the given Word.
998     // See Knuth, volume 4, section 7.3.1
999     func trailingZeroBits(x Word) int {
1000         // x & -x leaves only the right-most bit set in the
1001         // index of that bit. Since only a single bit is set
1002         // to the power of k. Multiplying by a power of two
1003         // left shifting, in this case by k bits. The de Br
1004         // such that all six bit, consecutive substrings are
1005         // Therefore, if we have a left shifted version of t
1006         // find by how many bits it was shifted by looking a
1007         // substring ended up at the top of the word.
1008         switch _W {
1009             case 32:
1010                 return int(deBruijn32Lookup[((x&-x)*deBruijn
1011             case 64:
1012                 return int(deBruijn64Lookup[((x&-x)*(deBruij
1013             default:
1014                 panic("Unknown word size")
1015         }
1016
1017         return 0
1018     }
1019
1020     // z = x << s
1021     func (z nat) shl(x nat, s uint) nat {
1022         m := len(x)
1023         if m == 0 {
1024             return z.make(0)
1025         }
1026         // m > 0
1027
1028         n := m + int(s/_W)
1029         z = z.make(n + 1)
1030         z[n] = shlVU(z[n-m:n], x, s%_W)
1031         z[0 : n-m].clear()

```

```

1032
1033         return z.norm()
1034     }
1035
1036     // z = x >> s
1037     func (z nat) shr(x nat, s uint) nat {
1038         m := len(x)
1039         n := m - int(s/_W)
1040         if n <= 0 {
1041             return z.make(0)
1042         }
1043         // n > 0
1044
1045         z = z.make(n)
1046         shrVU(z, x[m-n:], s%_W)
1047
1048         return z.norm()
1049     }
1050
1051     func (z nat) setBit(x nat, i uint, b uint) nat {
1052         j := int(i / _W)
1053         m := Word(1) << (i % _W)
1054         n := len(x)
1055         switch b {
1056         case 0:
1057             z = z.make(n)
1058             copy(z, x)
1059             if j >= n {
1060                 // no need to grow
1061                 return z
1062             }
1063             z[j] &^= m
1064             return z.norm()
1065         case 1:
1066             if j >= n {
1067                 z = z.make(j + 1)
1068                 z[n:].clear()
1069             } else {
1070                 z = z.make(n)
1071             }
1072             copy(z, x)
1073             z[j] |= m
1074             // no need to normalize
1075             return z
1076         }
1077         panic("set bit is not 0 or 1")
1078     }
1079
1080     func (z nat) bit(i uint) uint {

```

```

1081         j := int(i / _W)
1082         if j >= len(z) {
1083             return 0
1084         }
1085         return uint(z[j] >> (i % _W) & 1)
1086     }
1087
1088     func (z nat) and(x, y nat) nat {
1089         m := len(x)
1090         n := len(y)
1091         if m > n {
1092             m = n
1093         }
1094         // m <= n
1095
1096         z = z.make(m)
1097         for i := 0; i < m; i++ {
1098             z[i] = x[i] & y[i]
1099         }
1100
1101         return z.norm()
1102     }
1103
1104     func (z nat) andNot(x, y nat) nat {
1105         m := len(x)
1106         n := len(y)
1107         if n > m {
1108             n = m
1109         }
1110         // m >= n
1111
1112         z = z.make(m)
1113         for i := 0; i < n; i++ {
1114             z[i] = x[i] &^ y[i]
1115         }
1116         copy(z[n:m], x[n:m])
1117
1118         return z.norm()
1119     }
1120
1121     func (z nat) or(x, y nat) nat {
1122         m := len(x)
1123         n := len(y)
1124         s := x
1125         if m < n {
1126             n, m = m, n
1127             s = y
1128         }
1129         // m >= n
1130

```

```

1131     z = z.make(m)
1132     for i := 0; i < n; i++ {
1133         z[i] = x[i] | y[i]
1134     }
1135     copy(z[n:m], s[n:m])
1136
1137     return z.norm()
1138 }
1139
1140 func (z nat) xor(x, y nat) nat {
1141     m := len(x)
1142     n := len(y)
1143     s := x
1144     if m < n {
1145         n, m = m, n
1146         s = y
1147     }
1148     // m >= n
1149
1150     z = z.make(m)
1151     for i := 0; i < n; i++ {
1152         z[i] = x[i] ^ y[i]
1153     }
1154     copy(z[n:m], s[n:m])
1155
1156     return z.norm()
1157 }
1158
1159 // greaterThan returns true iff (x1<<_W + x2) > (y1<<_W + y2
1160 func greaterThan(x1, x2, y1, y2 Word) bool {
1161     return x1 > y1 || x1 == y1 && x2 > y2
1162 }
1163
1164 // modW returns x % d.
1165 func (x nat) modW(d Word) (r Word) {
1166     // TODO(agl): we don't actually need to store the q
1167     var q nat
1168     q = q.make(len(x))
1169     return divVWV(q, 0, x, d)
1170 }
1171
1172 // powersOfTwoDecompose finds q and k with x = q * 1<<k and
1173 func (x nat) powersOfTwoDecompose() (q nat, k int) {
1174     if len(x) == 0 {
1175         return x, 0
1176     }
1177
1178     // One of the words must be non-zero by definition,
1179     // so this loop will terminate with i < len(x), and

```

```

1180         // i is the number of 0 words.
1181         i := 0
1182         for x[i] == 0 {
1183             i++
1184         }
1185         n := trailingZeroBits(x[i]) // x[i] != 0
1186
1187         q = make(nat, len(x)-i)
1188         shrVU(q, x[i:], uint(n))
1189
1190         q = q.norm()
1191         k = i*_W + n
1192         return
1193     }
1194
1195     // random creates a random integer in [0..limit), using the
1196     // possible. n is the bit length of limit.
1197     func (z nat) random(rand *rand.Rand, limit nat, n int) nat {
1198         if alias(z, limit) {
1199             z = nil // z is an alias for limit - cannot
1200         }
1201         z = z.make(len(limit))
1202
1203         bitLengthOfMSW := uint(n % _W)
1204         if bitLengthOfMSW == 0 {
1205             bitLengthOfMSW = _W
1206         }
1207         mask := Word((1 << bitLengthOfMSW) - 1)
1208
1209         for {
1210             for i := range z {
1211                 switch _W {
1212                 case 32:
1213                     z[i] = Word(rand.Uint32())
1214                 case 64:
1215                     z[i] = Word(rand.Uint32()) |
1216                 }
1217             }
1218
1219             z[len(limit)-1] &= mask
1220
1221             if z.cmp(limit) < 0 {
1222                 break
1223             }
1224         }
1225
1226         return z.norm()
1227     }
1228

```

```

1229 // If m != nil, expNN calculates x**y mod m. Otherwise it ca
1230 // reuses the storage of z if possible.
1231 func (z nat) expNN(x, y, m nat) nat {
1232     if alias(z, x) || alias(z, y) {
1233         // We cannot allow in place modification of
1234         z = nil
1235     }
1236
1237     if len(y) == 0 {
1238         z = z.make(1)
1239         z[0] = 1
1240         return z
1241     }
1242
1243     if m != nil {
1244         // We likely end up being as long as the mod
1245         z = z.make(len(m))
1246     }
1247     z = z.set(x)
1248     v := y[len(y)-1]
1249     // It's invalid for the most significant word to be
1250     // will find a one bit.
1251     shift := leadingZeros(v) + 1
1252     v <<= shift
1253     var q nat
1254
1255     const mask = 1 << (_W - 1)
1256
1257     // We walk through the bits of the exponent one by o
1258     // see a bit, we square, thus doubling the power. If
1259     // we also multiply by x, thus adding one to the pow
1260
1261     w := _W - int(shift)
1262     for j := 0; j < w; j++ {
1263         z = z.mul(z, z)
1264
1265         if v&mask != 0 {
1266             z = z.mul(z, x)
1267         }
1268
1269         if m != nil {
1270             q, z = q.div(z, z, m)
1271         }
1272
1273         v <<= 1
1274     }
1275
1276     for i := len(y) - 2; i >= 0; i-- {
1277         v = y[i]
1278

```

```

1279         for j := 0; j < _W; j++ {
1280             z = z.mul(z, z)
1281
1282             if v&mask != 0 {
1283                 z = z.mul(z, x)
1284             }
1285
1286             if m != nil {
1287                 q, z = q.div(z, z, m)
1288             }
1289
1290             v <<= 1
1291         }
1292     }
1293
1294     return z.norm()
1295 }
1296
1297 // probablyPrime performs reps Miller-Rabin tests to check w
1298 // If it returns true, n is prime with probability 1 - 1/4^r
1299 // If it returns false, n is not prime.
1300 func (n nat) probablyPrime(reps int) bool {
1301     if len(n) == 0 {
1302         return false
1303     }
1304
1305     if len(n) == 1 {
1306         if n[0] < 2 {
1307             return false
1308         }
1309
1310         if n[0]%2 == 0 {
1311             return n[0] == 2
1312         }
1313
1314         // We have to exclude these cases because we
1315         // multiples of these numbers below.
1316         switch n[0] {
1317             case 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37
1318                 return true
1319         }
1320     }
1321
1322     const primesProduct32 = 0xC0CFD797 // Π {p ∈
1323     const primesProduct64 = 0xE221F97C30E94E1D // Π {p ∈
1324
1325     var r Word
1326     switch _W {
1327     case 32:

```

```

1328         r = n.modW(primesProduct32)
1329     case 64:
1330         r = n.modW(primesProduct64 & _M)
1331     default:
1332         panic("Unknown word size")
1333     }
1334
1335     if r%3 == 0 || r%5 == 0 || r%7 == 0 || r%11 == 0 ||
1336         r%13 == 0 || r%17 == 0 || r%19 == 0 || r%23
1337         return false
1338     }
1339
1340     if _W == 64 && (r%31 == 0 || r%37 == 0 || r%41 == 0
1341         r%43 == 0 || r%47 == 0 || r%53 == 0) {
1342         return false
1343     }
1344
1345     nm1 := nat(nil).sub(n, natOne)
1346     // 1<<k * q = nm1;
1347     q, k := nm1.powersOfTwoDecompose()
1348
1349     nm3 := nat(nil).sub(nm1, natTwo)
1350     rand := rand.New(rand.NewSource(int64(n[0])))
1351
1352     var x, y, quotient nat
1353     nm3Len := nm3.bitLen()
1354
1355     NextRandom:
1356         for i := 0; i < reps; i++ {
1357             x = x.random(rand, nm3, nm3Len)
1358             x = x.add(x, natTwo)
1359             y = y.expNN(x, q, n)
1360             if y.cmp(natOne) == 0 || y.cmp(nm1) == 0 {
1361                 continue
1362             }
1363             for j := 1; j < k; j++ {
1364                 y = y.mul(y, y)
1365                 quotient, y = quotient.div(y, y, n)
1366                 if y.cmp(nm1) == 0 {
1367                     continue NextRandom
1368                 }
1369                 if y.cmp(natOne) == 0 {
1370                     return false
1371                 }
1372             }
1373             return false
1374         }
1375
1376     return true

```

```

1377 }
1378
1379 // bytes writes the value of z into buf using big-endian enc
1380 // len(buf) must be >= len(z)*_S. The value of z is encoded
1381 // slice buf[i:]. The number i of unused bytes at the beginn
1382 // buf is returned as result.
1383 func (z nat) bytes(buf []byte) (i int) {
1384     i = len(buf)
1385     for _, d := range z {
1386         for j := 0; j < _S; j++ {
1387             i--
1388             buf[i] = byte(d)
1389             d >>= 8
1390         }
1391     }
1392
1393     for i < len(buf) && buf[i] == 0 {
1394         i++
1395     }
1396
1397     return
1398 }
1399
1400 // setBytes interprets buf as the bytes of a big-endian unsi
1401 // integer, sets z to that value, and returns z.
1402 func (z nat) setBytes(buf []byte) nat {
1403     z = z.make((len(buf) + _S - 1) / _S)
1404
1405     k := 0
1406     s := uint(0)
1407     var d Word
1408     for i := len(buf); i > 0; i-- {
1409         d |= Word(buf[i-1]) << s
1410         if s += 8; s == _S*8 {
1411             z[k] = d
1412             k++
1413             s = 0
1414             d = 0
1415         }
1416     }
1417     if k < len(z) {
1418         z[k] = d
1419     }
1420
1421     return z.norm()
1422 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/big/rat.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file implements multi-precision rational numbers.
6
7 package big
8
9 import (
10     "encoding/binary"
11     "errors"
12     "fmt"
13     "strings"
14 )
15
16 // A Rat represents a quotient a/b of arbitrary precision.
17 // The zero value for a Rat represents the value 0.
18 type Rat struct {
19     a Int
20     b nat // len(b) == 0 acts like b == 1
21 }
22
23 // NewRat creates a new Rat with numerator a and denominator
24 func NewRat(a, b int64) *Rat {
25     return new(Rat).SetFrac64(a, b)
26 }
27
28 // SetFrac sets z to a/b and returns z.
29 func (z *Rat) SetFrac(a, b *Int) *Rat {
30     z.a.neg = a.neg != b.neg
31     babs := b.abs
32     if len(babs) == 0 {
33         panic("division by zero")
34     }
35     if &z.a == b || alias(z.a.abs, babs) {
36         babs = nat(nil).set(babs) // make a copy
37     }
38     z.a.abs = z.a.abs.set(a.abs)
39     z.b = z.b.set(babs)
40     return z.norm()
41 }
42
43 // SetFrac64 sets z to a/b and returns z.
44 func (z *Rat) SetFrac64(a, b int64) *Rat {
```

```

45         z.a.SetInt64(a)
46         if b == 0 {
47             panic("division by zero")
48         }
49         if b < 0 {
50             b = -b
51             z.a.neg = !z.a.neg
52         }
53         z.b = z.b.setUint64(uint64(b))
54         return z.norm()
55     }
56
57     // SetInt sets z to x (by making a copy of x) and returns z.
58     func (z *Rat) SetInt(x *Int) *Rat {
59         z.a.Set(x)
60         z.b = z.b.make(0)
61         return z
62     }
63
64     // SetInt64 sets z to x and returns z.
65     func (z *Rat) SetInt64(x int64) *Rat {
66         z.a.SetInt64(x)
67         z.b = z.b.make(0)
68         return z
69     }
70
71     // Set sets z to x (by making a copy of x) and returns z.
72     func (z *Rat) Set(x *Rat) *Rat {
73         if z != x {
74             z.a.Set(&x.a)
75             z.b = z.b.set(x.b)
76         }
77         return z
78     }
79
80     // Abs sets z to |x| (the absolute value of x) and returns z
81     func (z *Rat) Abs(x *Rat) *Rat {
82         z.Set(x)
83         z.a.neg = false
84         return z
85     }
86
87     // Neg sets z to -x and returns z.
88     func (z *Rat) Neg(x *Rat) *Rat {
89         z.Set(x)
90         z.a.neg = len(z.a.abs) > 0 && !z.a.neg // 0 has no s
91         return z
92     }
93
94     // Inv sets z to 1/x and returns z.

```

```

95 func (z *Rat) Inv(x *Rat) *Rat {
96     if len(x.a.abs) == 0 {
97         panic("division by zero")
98     }
99     z.Set(x)
100    a := z.b
101    if len(a) == 0 {
102        a = a.setWord(1) // materialize numerator
103    }
104    b := z.a.abs
105    if b.cmp(natOne) == 0 {
106        b = b.make(0) // normalize denominator
107    }
108    z.a.abs, z.b = a, b // sign doesn't change
109    return z
110 }
111
112 // Sign returns:
113 //
114 //     -1 if x < 0
115 //     0  if x == 0
116 //     +1 if x > 0
117 //
118 func (x *Rat) Sign() int {
119     return x.a.Sign()
120 }
121
122 // IsInt returns true if the denominator of x is 1.
123 func (x *Rat) IsInt() bool {
124     return len(x.b) == 0 || x.b.cmp(natOne) == 0
125 }
126
127 // Num returns the numerator of x; it may be <= 0.
128 // The result is a reference to x's numerator; it
129 // may change if a new value is assigned to x.
130 func (x *Rat) Num() *Int {
131     return &x.a
132 }
133
134 // Denom returns the denominator of x; it is always > 0.
135 // The result is a reference to x's denominator; it
136 // may change if a new value is assigned to x.
137 func (x *Rat) Denom() *Int {
138     if len(x.b) == 0 {
139         return &Int{abs: nat{1}}
140     }
141     return &Int{abs: x.b}
142 }
143

```

```

144 func gcd(x, y nat) nat {
145     // Euclidean algorithm.
146     var a, b nat
147     a = a.set(x)
148     b = b.set(y)
149     for len(b) != 0 {
150         var q, r nat
151         _, r = q.div(r, a, b)
152         a = b
153         b = r
154     }
155     return a
156 }
157
158 func (z *Rat) norm() *Rat {
159     switch {
160     case len(z.a.abs) == 0:
161         // z == 0 - normalize sign and denominator
162         z.a.neg = false
163         z.b = z.b.make(0)
164     case len(z.b) == 0:
165         // z is normalized int - nothing to do
166     case z.b.cmp(natOne) == 0:
167         // z is int - normalize denominator
168         z.b = z.b.make(0)
169     default:
170         if f := gcd(z.a.abs, z.b); f.cmp(natOne) !=
171             z.a.abs, _ = z.a.abs.div(nil, z.a.ab
172             z.b, _ = z.b.div(nil, z.b, f)
173         }
174     }
175     return z
176 }
177
178 // mulDenom sets z to the denominator product x*y (by taking
179 // account that 0 values for x or y must be interpreted as 1
180 // returns z.
181 func mulDenom(z, x, y nat) nat {
182     switch {
183     case len(x) == 0:
184         return z.set(y)
185     case len(y) == 0:
186         return z.set(x)
187     }
188     return z.mul(x, y)
189 }
190
191 // scaleDenom computes x*f.
192 // If f == 0 (zero value of denominator), the result is (a c

```

```

193 func scaleDenom(x *Int, f nat) *Int {
194     var z Int
195     if len(f) == 0 {
196         return z.Set(x)
197     }
198     z.abs = z.abs.mul(x.abs, f)
199     z.neg = x.neg
200     return &z
201 }
202
203 // Cmp compares x and y and returns:
204 //
205 //   -1 if x < y
206 //    0 if x == y
207 //   +1 if x > y
208 //
209 func (x *Rat) Cmp(y *Rat) int {
210     return scaleDenom(&x.a, y.b).Cmp(scaleDenom(&y.a, x.
211 }
212
213 // Add sets z to the sum x+y and returns z.
214 func (z *Rat) Add(x, y *Rat) *Rat {
215     a1 := scaleDenom(&x.a, y.b)
216     a2 := scaleDenom(&y.a, x.b)
217     z.a.Add(a1, a2)
218     z.b = mulDenom(z.b, x.b, y.b)
219     return z.norm()
220 }
221
222 // Sub sets z to the difference x-y and returns z.
223 func (z *Rat) Sub(x, y *Rat) *Rat {
224     a1 := scaleDenom(&x.a, y.b)
225     a2 := scaleDenom(&y.a, x.b)
226     z.a.Sub(a1, a2)
227     z.b = mulDenom(z.b, x.b, y.b)
228     return z.norm()
229 }
230
231 // Mul sets z to the product x*y and returns z.
232 func (z *Rat) Mul(x, y *Rat) *Rat {
233     z.a.Mul(&x.a, &y.a)
234     z.b = mulDenom(z.b, x.b, y.b)
235     return z.norm()
236 }
237
238 // Quo sets z to the quotient x/y and returns z.
239 // If y == 0, a division-by-zero run-time panic occurs.
240 func (z *Rat) Quo(x, y *Rat) *Rat {
241     if len(y.a.abs) == 0 {
242         panic("division by zero")

```

```

243     }
244     a := scaleDenom(&x.a, y.b)
245     b := scaleDenom(&y.a, x.b)
246     z.a.abs = a.abs
247     z.b = b.abs
248     z.a.neg = a.neg != b.neg
249     return z.norm()
250 }
251
252 func ratTok(ch rune) bool {
253     return strings.IndexRune("+-/0123456789.eE", ch) >=
254 }
255
256 // Scan is a support routine for fmt.Scanner. It accepts the
257 // 'e', 'E', 'f', 'F', 'g', 'G', and 'v'. All formats are eq
258 func (z *Rat) Scan(s fmt.ScanState, ch rune) error {
259     tok, err := s.Token(true, ratTok)
260     if err != nil {
261         return err
262     }
263     if strings.IndexRune("efgEFGv", ch) < 0 {
264         return errors.New("Rat.Scan: invalid verb")
265     }
266     if _, ok := z.SetString(string(tok)); !ok {
267         return errors.New("Rat.Scan: invalid syntax")
268     }
269     return nil
270 }
271
272 // SetString sets z to the value of s and returns z and a bo
273 // success. s can be given as a fraction "a/b" or as a float
274 // optionally followed by an exponent. If the operation fail
275 // z is undefined but the returned value is nil.
276 func (z *Rat) SetString(s string) (*Rat, bool) {
277     if len(s) == 0 {
278         return nil, false
279     }
280
281     // check for a quotient
282     sep := strings.Index(s, "/")
283     if sep >= 0 {
284         if _, ok := z.a.SetString(s[0:sep], 10); !ok
285             return nil, false
286     }
287     s = s[sep+1:]
288     var err error
289     if z.b, _, err = z.b.scan(strings.NewReader(
290         return nil, false
291     }

```

```

292         return z.norm(), true
293     }
294
295     // check for a decimal point
296     sep = strings.Index(s, ".")
297     // check for an exponent
298     e := strings.IndexAny(s, "eE")
299     var exp Int
300     if e >= 0 {
301         if e < sep {
302             // The E must come after the decimal
303             return nil, false
304         }
305         if _, ok := exp.SetString(s[e+1:], 10); !ok
306             return nil, false
307     }
308     s = s[0:e]
309 }
310 if sep >= 0 {
311     s = s[0:sep] + s[sep+1:]
312     exp.Sub(&exp, NewInt(int64(len(s)-sep)))
313 }
314
315 if _, ok := z.a.SetString(s, 10); !ok {
316     return nil, false
317 }
318 powTen := nat(nil).expNN(natTen, exp.abs, nil)
319 if exp.neg {
320     z.b = powTen
321     z.norm()
322 } else {
323     z.a.abs = z.a.abs.mul(z.a.abs, powTen)
324     z.b = z.b.make(0)
325 }
326
327 return z, true
328 }
329
330 // String returns a string representation of z in the form "
331 func (x *Rat) String() string {
332     s := "/1"
333     if len(x.b) != 0 {
334         s = "/" + x.b.decimalString()
335     }
336     return x.a.String() + s
337 }
338
339 // RatString returns a string representation of z in the for
340 // and in the form "a" if b == 1.

```

```

341 func (x *Rat) RatString() string {
342     if x.IsInt() {
343         return x.a.String()
344     }
345     return x.String()
346 }
347
348 // FloatString returns a string representation of z in decim
349 // digits of precision after the decimal point and the last
350 func (x *Rat) FloatString(prec int) string {
351     if x.IsInt() {
352         s := x.a.String()
353         if prec > 0 {
354             s += "." + strings.Repeat("0", prec)
355         }
356         return s
357     }
358     // x.b != 0
359
360     q, r := nat(nil).div(nat(nil), x.a.abs, x.b)
361
362     p := natOne
363     if prec > 0 {
364         p = nat(nil).expNN(natTen, nat(nil).setUint6
365     }
366
367     r = r.mul(r, p)
368     r, r2 := r.div(nat(nil), r, x.b)
369
370     // see if we need to round up
371     r2 = r2.add(r2, r2)
372     if x.b.cmp(r2) <= 0 {
373         r = r.add(r, natOne)
374         if r.cmp(p) >= 0 {
375             q = nat(nil).add(q, natOne)
376             r = nat(nil).sub(r, p)
377         }
378     }
379
380     s := q.decimalString()
381     if x.a.neg {
382         s = "-" + s
383     }
384
385     if prec > 0 {
386         rs := r.decimalString()
387         leadingZeros := prec - len(rs)
388         s += "." + strings.Repeat("0", leadingZeros)
389     }
390 }

```

```

391         return s
392     }
393
394 // Gob codec version. Permits backward-compatible changes to
395 const ratGobVersion byte = 1
396
397 // GobEncode implements the gob.GobEncoder interface.
398 func (x *Rat) GobEncode() ([]byte, error) {
399     buf := make([]byte, 1+4+(len(x.a.abs)+len(x.b))*_S)
400     i := x.b.bytes(buf)
401     j := x.a.abs.bytes(buf[0:i])
402     n := i - j
403     if int(uint32(n)) != n {
404         // this should never happen
405         return nil, errors.New("Rat.GobEncode: numer
406     }
407     binary.BigEndian.PutUint32(buf[j-4:j], uint32(n))
408     j -= 1 + 4
409     b := ratGobVersion << 1 // make space for sign bit
410     if x.a.neg {
411         b |= 1
412     }
413     buf[j] = b
414     return buf[j:], nil
415 }
416
417 // GobDecode implements the gob.GobDecoder interface.
418 func (z *Rat) GobDecode(buf []byte) error {
419     if len(buf) == 0 {
420         return errors.New("Rat.GobDecode: no data")
421     }
422     b := buf[0]
423     if b>>1 != ratGobVersion {
424         return errors.New(fmt.Sprintf("Rat.GobDecode
425     }
426     const j = 1 + 4
427     i := j + binary.BigEndian.Uint32(buf[j-4:j])
428     z.a.neg = b&1 != 0
429     z.a.abs = z.a.abs.setBytes(buf[j:i])
430     z.b = z.b.setBytes(buf[i:])
431     return nil
432 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/abs.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package cmplx provides basic constants and mathematical f
6 // complex numbers.
7 package cmplx
8
9 import "math"
10
11 // Abs returns the absolute value (also called the modulus)
12 func Abs(x complex128) float64 { return math.Hypot(real(x),
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/asin.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // The original C code, the long comment, and the constants
10 // below are from http://netlib.sandia.gov/cephes/c9x-comple
11 // The go code is a simplified version of the original C.
12 //
13 // Cephes Math Library Release 2.8: June, 2000
14 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
15 //
16 // The readme file at http://netlib.sandia.gov/cephes/ says:
17 // Some software in this archive may be from the book Me
18 // Programs for Mathematical Functions (Prentice-Hall or Si
19 // International, 1989) or from the Cephes Mathematical Libr
20 // commercial product. In either event, it is copyrighted by
21 // What you see here may be used freely but it comes with no
22 // guarantee.
23 //
24 // The two known misprints in the book are repaired here i
25 // source listings for the gamma function and the incomplete
26 // integral.
27 //
28 // Stephen L. Moshier
29 // moshier@na-net.ornl.gov
30
31 // Complex circular arc sine
32 //
33 // DESCRIPTION:
34 //
35 // Inverse complex sine:
36 //
37 //  $w = -i \operatorname{clog}(iz + \operatorname{csqrt}(1 - z^2))$ .
38 //
39 //  $\operatorname{casin}(z) = -i \operatorname{casinh}(iz)$ 
40 //
41 // ACCURACY:
```

```

42 //
43 //
44 // arithmetic domain # trials peak rms
45 // DEC -10,+10 10100 2.1e-15 3.4e-16
46 // IEEE -10,+10 30000 2.2e-14 2.7e-15
47 // Larger relative error can be observed for z near zero.
48 // Also tested by csin(casin(z)) = z.
49
50 // Asin returns the inverse sine of x.
51 func Asin(x complex128) complex128 {
52     if imag(x) == 0 {
53         if math.Abs(real(x)) > 1 {
54             return complex(math.Pi/2, 0) // DOMA
55         }
56         return complex(math.Asin(real(x)), 0)
57     }
58     ct := complex(-imag(x), real(x)) // i * x
59     xx := x * x
60     x1 := complex(1-real(xx), -imag(xx)) // 1 - x*x
61     x2 := Sqrt(x1) // x2 = sqrt(1
62     w := Log(ct + x2)
63     return complex(imag(w), -real(w)) // -i * w
64 }
65
66 // Asinh returns the inverse hyperbolic sine of x.
67 func Asinh(x complex128) complex128 {
68     // TODO check range
69     if imag(x) == 0 {
70         if math.Abs(real(x)) > 1 {
71             return complex(math.Pi/2, 0) // DOMA
72         }
73         return complex(math.Asinh(real(x)), 0)
74     }
75     xx := x * x
76     x1 := complex(1+real(xx), imag(xx)) // 1 + x*x
77     return Log(x + Sqrt(x1)) // log(x + sqrt(
78 }
79
80 // Complex circular arc cosine
81 //
82 // DESCRIPTION:
83 //
84 //  $w = \arccos z = \text{PI}/2 - \arcsin z.$ 
85 //
86 // ACCURACY:
87 //
88 //
89 // arithmetic domain # trials peak rms
90 // DEC -10,+10 5200 1.6e-15 2.8e-16
91 // IEEE -10,+10 30000 1.8e-14 2.2e-15

```

```

92
93 // Acos returns the inverse cosine of x.
94 func Acos(x complex128) complex128 {
95     w := Asin(x)
96     return complex(math.Pi/2-real(w), -imag(w))
97 }
98
99 // Acosh returns the inverse hyperbolic cosine of x.
100 func Acosh(x complex128) complex128 {
101     w := Acos(x)
102     if imag(w) <= 0 {
103         return complex(-imag(w), real(w)) // i * w
104     }
105     return complex(imag(w), -real(w)) // -i * w
106 }
107
108 // Complex circular arc tangent
109 //
110 // DESCRIPTION:
111 //
112 // If
113 //     z = x + iy,
114 //
115 // then
116 //
117 //  $\text{Re } w = \frac{1}{2} \arctan\left(\frac{2x}{1 - x^2 - y^2}\right) + k \text{ PI}$ 
118 //
119 //
120 //
121 //
122 //  $\text{Im } w = \frac{1}{4} \log\left(\frac{x^2 + (y+1)^2}{x^2 + (y-1)^2}\right)$ 
123 //
124 //
125 //
126 //
127 // Where k is an arbitrary integer.
128 //
129 //  $\text{catan}(z) = -i \text{catanh}(iz)$ .
130 //
131 // ACCURACY:
132 //
133 //
134 // 




139 // The check  $\text{catan}(\text{ctan}(z)) = z$ , with  $|x|$  and  $|y| < \text{PI}/2$ 
140 // had peak relative error 1.5e-16, rms relative error
141 // 2.9e-17. See also clog().
142

```

```

141 // Atan returns the inverse tangent of x.
142 func Atan(x complex128) complex128 {
143     if real(x) == 0 && imag(x) > 1 {
144         return NaN()
145     }
146
147     x2 := real(x) * real(x)
148     a := 1 - x2 - imag(x)*imag(x)
149     if a == 0 {
150         return NaN()
151     }
152     t := 0.5 * math.Atan2(2*real(x), a)
153     w := reducePi(t)
154
155     t = imag(x) - 1
156     b := x2 + t*t
157     if b == 0 {
158         return NaN()
159     }
160     t = imag(x) + 1
161     c := (x2 + t*t) / b
162     return complex(w, 0.25*math.Log(c))
163 }
164
165 // Atanh returns the inverse hyperbolic tangent of x.
166 func Atanh(x complex128) complex128 {
167     z := complex(-imag(x), real(x)) // z = i * x
168     z = Atan(z)
169     return complex(imag(z), -real(z)) // z = -i * z
170 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/conj.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.  
2 // Use of this source code is governed by a BSD-style  
3 // license that can be found in the LICENSE file.  
4  
5 package cmplx  
6  
7 // Conj returns the complex conjugate of x.  
8 func Conj(x complex128) complex128 { return complex(real(x),
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/exp.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // The original C code, the long comment, and the constants
10 // below are from http://netlib.sandia.gov/cephes/c9x-comple
11 // The go code is a simplified version of the original C.
12 //
13 // Cephes Math Library Release 2.8: June, 2000
14 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
15 //
16 // The readme file at http://netlib.sandia.gov/cephes/ says:
17 // Some software in this archive may be from the book Me
18 // Programs for Mathematical Functions (Prentice-Hall or Si
19 // International, 1989) or from the Cephes Mathematical Libr
20 // commercial product. In either event, it is copyrighted by
21 // What you see here may be used freely but it comes with no
22 // guarantee.
23 //
24 // The two known misprints in the book are repaired here i
25 // source listings for the gamma function and the incomplete
26 // integral.
27 //
28 // Stephen L. Moshier
29 // moshier@na-net.ornl.gov
30
31 // Complex exponential function
32 //
33 // DESCRIPTION:
34 //
35 // Returns the complex exponential of the complex argument z
36 //
37 // If
38 //     z = x + iy,
39 //     r = exp(x),
40 // then
41 //     w = r cos y + i r sin y.
```

```

42 //
43 // ACCURACY:
44 //
45 //                               Relative error:
46 // arithmetic    domain    # trials    peak        rms
47 //    DEC        -10,+10    8700        3.7e-17     1.1e-17
48 //    IEEE       -10,+10    30000       3.0e-16     8.7e-17
49
50 // Exp returns e**x, the base-e exponential of x.
51 func Exp(x complex128) complex128 {
52     r := math.Exp(real(x))
53     s, c := math.Sincos(imag(x))
54     return complex(r*c, r*s)
55 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/isinf.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // IsInf returns true if either real(x) or imag(x) is an inf
10 func IsInf(x complex128) bool {
11     if math.IsInf(real(x), 0) || math.IsInf(imag(x), 0)
12         return true
13     }
14     return false
15 }
16
17 // Inf returns a complex infinity, complex(+Inf, +Inf).
18 func Inf() complex128 {
19     inf := math.Inf(1)
20     return complex(inf, inf)
21 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/isnan.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // IsNaN returns true if either real(x) or imag(x) is NaN
10 // and neither is an infinity.
11 func IsNaN(x complex128) bool {
12     switch {
13     case math.IsInf(real(x), 0) || math.IsInf(imag(x), 0):
14         return false
15     case math.IsNaN(real(x)) || math.IsNaN(imag(x)):
16         return true
17     }
18     return false
19 }
20
21 // NaN returns a complex ``not-a-number'' value.
22 func NaN() complex128 {
23     nan := math.NaN()
24     return complex(nan, nan)
25 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/cmplx/log.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // The original C code, the long comment, and the constants
10 // below are from http://netlib.sandia.gov/cephes/c9x-comple
11 // The go code is a simplified version of the original C.
12 //
13 // Cephes Math Library Release 2.8: June, 2000
14 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
15 //
16 // The readme file at http://netlib.sandia.gov/cephes/ says:
17 //     Some software in this archive may be from the book Me
18 // Programs for Mathematical Functions (Prentice-Hall or Si
19 // International, 1989) or from the Cephes Mathematical Libr
20 // commercial product. In either event, it is copyrighted by
21 // What you see here may be used freely but it comes with no
22 // guarantee.
23 //
24 //     The two known misprints in the book are repaired here i
25 // source listings for the gamma function and the incomplete
26 // integral.
27 //
28 //     Stephen L. Moshier
29 //     moshier@na-net.ornl.gov
30
31 // Complex natural logarithm
32 //
33 // DESCRIPTION:
34 //
35 // Returns complex logarithm to the base e (2.718...) of
36 // the complex argument z.
37 //
38 // If
39 //      $z = x + iy$ ,  $r = \sqrt{x^2 + y^2}$ ,
40 // then
41 //      $w = \log(r) + i \arctan(y/x)$ .
42 //
43 // The arctangent ranges from -PI to +PI.
44 //
```

```

45 // ACCURACY:
46 //
47 //
48 // arithmetic    domain    # trials    peak    rms
49 //   DEC        -10,+10    7000       8.5e-17  1.9e-17
50 //   IEEE       -10,+10    30000      5.0e-15  1.1e-16
51 //
52 // Larger relative error can be observed for z near 1 +i0.
53 // In IEEE arithmetic the peak absolute error is 5.2e-16, rm
54 // absolute error 1.0e-16.
55
56 // Log returns the natural logarithm of x.
57 func Log(x complex128) complex128 {
58     return complex(math.Log(Abs(x)), Phase(x))
59 }
60
61 // Log10 returns the decimal logarithm of x.
62 func Log10(x complex128) complex128 {
63     return math.Log10E * Log(x)
64 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/phase.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // Phase returns the phase (also called the argument) of x.
10 // The returned value is in the range [-Pi, Pi].
11 func Phase(x complex128) float64 { return math.Atan2(imag(x)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/polar.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 // Polar returns the absolute value r and phase  $\theta$  of x,
8 // such that  $x = r * e^{i\theta}$ .
9 // The phase is in the range  $[-\pi, \pi]$ .
10 func Polar(x complex128) (r,  $\theta$  float64) {
11     return Abs(x), Phase(x)
12 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/pow.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // The original C code, the long comment, and the constants
10 // below are from http://netlib.sandia.gov/cephes/c9x-comple
11 // The go code is a simplified version of the original C.
12 //
13 // Cephes Math Library Release 2.8: June, 2000
14 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
15 //
16 // The readme file at http://netlib.sandia.gov/cephes/ says:
17 // Some software in this archive may be from the book Me
18 // Programs for Mathematical Functions (Prentice-Hall or Si
19 // International, 1989) or from the Cephes Mathematical Libr
20 // commercial product. In either event, it is copyrighted by
21 // What you see here may be used freely but it comes with no
22 // guarantee.
23 //
24 // The two known misprints in the book are repaired here i
25 // source listings for the gamma function and the incomplete
26 // integral.
27 //
28 // Stephen L. Moshier
29 // moshier@na-net.ornl.gov
30
31 // Complex power function
32 //
33 // DESCRIPTION:
34 //
35 // Raises complex A to the complex Zth power.
36 // Definition is per AMS55 # 4.2.8,
37 // analytically equivalent to  $\text{cpow}(a,z) = \text{cexp}(z \text{clog}(a))$ .
38 //
39 // ACCURACY:
40 //
41 // Relative error:
```

```

42 // arithmetic    domain    # trials    peak    rms
43 //    IEEE      -10,+10    30000    9.4e-15  1.5e-15
44
45 // Pow returns x**y, the base-x exponential of y.
46 func Pow(x, y complex128) complex128 {
47     modulus := Abs(x)
48     if modulus == 0 {
49         return complex(0, 0)
50     }
51     r := math.Pow(modulus, real(y))
52     arg := Phase(x)
53     theta := real(y) * arg
54     if imag(y) != 0 {
55         r *= math.Exp(-imag(y) * arg)
56         theta += imag(y) * math.Log(modulus)
57     }
58     s, c := math.Sincos(theta)
59     return complex(r*c, r*s)
60 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/rect.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // Rect returns the complex number x with polar coordinates
10 func Rect(r, θ float64) complex128 {
11     s, c := math.Sincos(θ)
12     return complex(r*c, r*s)
13 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/cmplx/sin.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // The original C code, the long comment, and the constants
10 // below are from http://netlib.sandia.gov/cephes/c9x-comple
11 // The go code is a simplified version of the original C.
12 //
13 // Cephes Math Library Release 2.8: June, 2000
14 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
15 //
16 // The readme file at http://netlib.sandia.gov/cephes/ says:
17 //     Some software in this archive may be from the book Me
18 // Programs for Mathematical Functions (Prentice-Hall or Si
19 // International, 1989) or from the Cephes Mathematical Libr
20 // commercial product. In either event, it is copyrighted by
21 // What you see here may be used freely but it comes with no
22 // guarantee.
23 //
24 //     The two known misprints in the book are repaired here i
25 // source listings for the gamma function and the incomplete
26 // integral.
27 //
28 //     Stephen L. Moshier
29 //     moshier@na-net.ornl.gov
30
31 // Complex circular sine
32 //
33 // DESCRIPTION:
34 //
35 // If
36 //      $z = x + iy,$ 
37 //
38 // then
39 //
40 //      $w = \sin x \cosh y + i \cos x \sinh y.$ 
41 //
42 //  $\text{csin}(z) = -i \text{csinh}(iz).$ 
43 //
44 // ACCURACY:
```

```

45 //
46 //
47 // arithmetic domain # trials peak rms
48 // DEC -10,+10 8400 5.3e-17 1.3e-17
49 // IEEE -10,+10 30000 3.8e-16 1.0e-16
50 // Also tested by csin(casin(z)) = z.
51
52 // Sin returns the sine of x.
53 func Sin(x complex128) complex128 {
54     s, c := math.Sincos(real(x))
55     sh, ch := sinhcosh(imag(x))
56     return complex(s*ch, c*sh)
57 }
58
59 // Complex hyperbolic sine
60 //
61 // DESCRIPTION:
62 //
63 // csinh z = (cexp(z) - cexp(-z))/2
64 //          = sinh x * cos y + i cosh x * sin y .
65 //
66 // ACCURACY:
67 //
68 //
69 // arithmetic domain # trials peak rms
70 // IEEE -10,+10 30000 3.1e-16 8.2e-17
71
72 // Sinh returns the hyperbolic sine of x.
73 func Sinh(x complex128) complex128 {
74     s, c := math.Sincos(imag(x))
75     sh, ch := sinhcosh(real(x))
76     return complex(c*sh, s*ch)
77 }
78
79 // Complex circular cosine
80 //
81 // DESCRIPTION:
82 //
83 // If
84 //     z = x + iy,
85 //
86 // then
87 //
88 //     w = cos x cosh y - i sin x sinh y.
89 //
90 // ACCURACY:
91 //
92 //
93 // arithmetic domain # trials peak rms
94 // DEC -10,+10 8400 4.5e-17 1.3e-17

```

```

95 // IEEE -10,+10 30000 3.8e-16 1.0e-16
96
97 // Cos returns the cosine of x.
98 func Cos(x complex128) complex128 {
99     s, c := math.Sincos(real(x))
100     sh, ch := sinhcosh(imag(x))
101     return complex(c*ch, -s*sh)
102 }
103
104 // Complex hyperbolic cosine
105 //
106 // DESCRIPTION:
107 //
108 // ccosh(z) = cosh x cos y + i sinh x sin y .
109 //
110 // ACCURACY:
111 //
112 // Relative error:
113 // arithmetic domain # trials peak rms
114 // IEEE -10,+10 30000 2.9e-16 8.1e-17
115
116 // Cosh returns the hyperbolic cosine of x.
117 func Cosh(x complex128) complex128 {
118     s, c := math.Sincos(imag(x))
119     sh, ch := sinhcosh(real(x))
120     return complex(c*ch, s*sh)
121 }
122
123 // calculate sinh and cosh
124 func sinhcosh(x float64) (sh, ch float64) {
125     if math.Abs(x) <= 0.5 {
126         return math.Sinh(x), math.Cosh(x)
127     }
128     e := math.Exp(x)
129     ei := 0.5 / e
130     e *= 0.5
131     return e - ei, e + ei
132 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/cmplx/sqrt.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // The original C code, the long comment, and the constants
10 // below are from http://netlib.sandia.gov/cephes/c9x-comple
11 // The go code is a simplified version of the original C.
12 //
13 // Cephes Math Library Release 2.8: June, 2000
14 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
15 //
16 // The readme file at http://netlib.sandia.gov/cephes/ says:
17 // Some software in this archive may be from the book Me
18 // Programs for Mathematical Functions (Prentice-Hall or Si
19 // International, 1989) or from the Cephes Mathematical Libr
20 // commercial product. In either event, it is copyrighted by
21 // What you see here may be used freely but it comes with no
22 // guarantee.
23 //
24 // The two known misprints in the book are repaired here i
25 // source listings for the gamma function and the incomplete
26 // integral.
27 //
28 // Stephen L. Moshier
29 // moshier@na-net.ornl.gov
30
31 // Complex square root
32 //
33 // DESCRIPTION:
34 //
35 // If  $z = x + iy$ ,  $r = |z|$ , then
36 //
37 // 
$$\operatorname{Re} w = \left[ \frac{r + x}{2} \right]^{1/2},$$

38 //
39 //
40 // 
$$\operatorname{Im} w = \left[ \frac{r - x}{2} \right]^{1/2}.$$

41
```

```

42 //
43 // Cancellation error in r-x or r+x is avoided by using the
44 // identity  $2 \operatorname{Re} w \operatorname{Im} w = y$ .
45 //
46 // Note that -w is also a square root of z. The root chosen
47 // is always in the right half plane and  $\operatorname{Im} w$  has the same s
48 //
49 // ACCURACY:
50 //
51 //                               Relative error:
52 // arithmetic   domain   # trials   peak           rms
53 //   DEC        -10,+10   25000     3.2e-17        9.6e-18
54 //   IEEE       -10,+10  1,000,000 2.9e-16        6.1e-17
55
56 // Sqrt returns the square root of x.
57 func Sqrt(x complex128) complex128 {
58     if imag(x) == 0 {
59         if real(x) == 0 {
60             return complex(0, 0)
61         }
62         if real(x) < 0 {
63             return complex(0, math.Sqrt(-real(x)))
64         }
65         return complex(math.Sqrt(real(x)), 0)
66     }
67     if real(x) == 0 {
68         if imag(x) < 0 {
69             r := math.Sqrt(-0.5 * imag(x))
70             return complex(r, -r)
71         }
72         r := math.Sqrt(0.5 * imag(x))
73         return complex(r, r)
74     }
75     a := real(x)
76     b := imag(x)
77     var scale float64
78     // Rescale to avoid internal overflow or underflow.
79     if math.Abs(a) > 4 || math.Abs(b) > 4 {
80         a *= 0.25
81         b *= 0.25
82         scale = 2
83     } else {
84         a *= 1.8014398509481984e16 // 2**54
85         b *= 1.8014398509481984e16
86         scale = 7.450580596923828125e-9 // 2**-27
87     }
88     r := math.Hypot(a, b)
89     var t float64
90     if a > 0 {
91         t = math.Sqrt(0.5*r + 0.5*a)

```

```
92         r = scale * math.Abs((0.5*b)/t)
93         t *= scale
94     } else {
95         r = math.Sqrt(0.5*r - 0.5*a)
96         t = scale * math.Abs((0.5*b)/r)
97         r *= scale
98     }
99     if b < 0 {
100         return complex(t, -r)
101     }
102     return complex(t, r)
103 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/cmplx/tan.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cmplx
6
7 import "math"
8
9 // The original C code, the long comment, and the constants
10 // below are from http://netlib.sandia.gov/cephes/c9x-comple
11 // The go code is a simplified version of the original C.
12 //
13 // Cephes Math Library Release 2.8: June, 2000
14 // Copyright 1984, 1987, 1989, 1992, 2000 by Stephen L. Mosh
15 //
16 // The readme file at http://netlib.sandia.gov/cephes/ says:
17 //     Some software in this archive may be from the book Me
18 // Programs for Mathematical Functions (Prentice-Hall or Si
19 // International, 1989) or from the Cephes Mathematical Libr
20 // commercial product. In either event, it is copyrighted by
21 // What you see here may be used freely but it comes with no
22 // guarantee.
23 //
24 //     The two known misprints in the book are repaired here i
25 // source listings for the gamma function and the incomplete
26 // integral.
27 //
28 //     Stephen L. Moshier
29 //     moshier@na-net.ornl.gov
30
31 // Complex circular tangent
32 //
33 // DESCRIPTION:
34 //
35 // If
36 //      $z = x + iy,$ 
37 //
38 // then
39 //
40 //      $w = \frac{\sin 2x + i \sinh 2y}{\cos 2x + \cosh 2y}.$ 
41 //
42 //
43 //
44 // On the real axis the denominator is zero at odd multiples
```

```

45 // of PI/2. The denominator is evaluated by its Taylor
46 // series near these points.
47 //
48 // ctan(z) = -i ctanh(iz).
49 //
50 // ACCURACY:
51 //
52 //                               Relative error:
53 // arithmetic   domain   # trials   peak           rms
54 //   DEC        -10,+10   5200      7.1e-17        1.6e-17
55 //   IEEE       -10,+10  30000     7.2e-16        1.2e-16
56 // Also tested by ctan * ccot = 1 and catan(ctan(z)) = z.
57
58 // Tan returns the tangent of x.
59 func Tan(x complex128) complex128 {
60     d := math.Cos(2*real(x)) + math.Cosh(2*imag(x))
61     if math.Abs(d) < 0.25 {
62         d = tanSeries(x)
63     }
64     if d == 0 {
65         return Inf()
66     }
67     return complex(math.Sin(2*real(x))/d, math.Sinh(2*im
68 }
69
70 // Complex hyperbolic tangent
71 //
72 // DESCRIPTION:
73 //
74 // tanh z = (sinh 2x + i sin 2y) / (cosh 2x + cos 2y) .
75 //
76 // ACCURACY:
77 //
78 //                               Relative error:
79 // arithmetic   domain   # trials   peak           rms
80 //   IEEE       -10,+10  30000     1.7e-14        2.4e-16
81
82 // Tanh returns the hyperbolic tangent of x.
83 func Tanh(x complex128) complex128 {
84     d := math.Cosh(2*real(x)) + math.Cos(2*imag(x))
85     if d == 0 {
86         return Inf()
87     }
88     return complex(math.Sinh(2*real(x))/d, math.Sin(2*im
89 }
90
91 // Program to subtract nearest integer multiple of PI
92 func reducePi(x float64) float64 {
93     const (
94         // extended precision value of PI:

```

```

95             DP1 = 3.14159265160560607910E0    // ?? 0x400
96             DP2 = 1.98418714791870343106E-9    // ?? 0x3e2
97             DP3 = 1.14423774522196636802E-17    // ?? 0x3c6
98         )
99         t := x / math.Pi
100        if t >= 0 {
101            t += 0.5
102        } else {
103            t -= 0.5
104        }
105        t = float64(int64(t)) // int64(t) = the multiple
106        return ((x - t*DP1) - t*DP2) - t*DP3
107    }
108
109    // Taylor series expansion for cosh(2y) - cos(2x)
110    func tanSeries(z complex128) float64 {
111        const MACHEP = 1.0 / (1 << 53)
112        x := math.Abs(2 * real(z))
113        y := math.Abs(2 * imag(z))
114        x = reducePi(x)
115        x = x * x
116        y = y * y
117        x2 := 1.0
118        y2 := 1.0
119        f := 1.0
120        rn := 0.0
121        d := 0.0
122        for {
123            rn += 1
124            f *= rn
125            rn += 1
126            f *= rn
127            x2 *= x
128            y2 *= y
129            t := y2 + x2
130            t /= f
131            d += t
132
133            rn += 1
134            f *= rn
135            rn += 1
136            f *= rn
137            x2 *= x
138            y2 *= y
139            t = y2 - x2
140            t /= f
141            d += t
142            if math.Abs(t/d) <= MACHEP {
143                break

```

```

144         }
145     }
146     return d
147 }
148
149 // Complex circular cotangent
150 //
151 // DESCRIPTION:
152 //
153 // If
154 //     z = x + iy,
155 //
156 // then
157 //
158 //     sin 2x - i sinh 2y
159 //     w = -----
160 //     cosh 2y - cos 2x
161 //
162 // On the real axis, the denominator has zeros at even
163 // multiples of PI/2. Near these points it is evaluated
164 // by a Taylor series.
165 //
166 // ACCURACY:
167 //
168 //                                     Relative error:
169 // arithmetic   domain   # trials   peak           rms
170 //    DEC        -10,+10   3000      6.5e-17        1.6e-17
171 //    IEEE       -10,+10  30000     9.2e-16        1.2e-16
172 // Also tested by ctan * ccot = 1 + i0.
173
174 // Cot returns the cotangent of x.
175 func Cot(x complex128) complex128 {
176     d := math.Cosh(2*imag(x)) - math.Cos(2*real(x))
177     if math.Abs(d) < 0.25 {
178         d = tanSeries(x)
179     }
180     if d == 0 {
181         return Inf()
182     }
183     return complex(math.Sin(2*real(x))/d, -math.Sinh(2*i
184 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/rand/exp.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package rand
6
7 import (
8     "math"
9 )
10
11 /*
12  * Exponential distribution
13  *
14  * See "The Ziggurat Method for Generating Random Variables"
15  * (Marsaglia & Tsang, 2000)
16  * http://www.jstatsoft.org/v05/i08/paper [pdf]
17  */
18
19 const (
20     re = 7.69711747013104972
21 )
22
23 // ExpFloat64 returns an exponentially distributed float64 i
24 // (0, +math.MaxFloat64] with an exponential distribution wh
25 // (lambda) is 1 and whose mean is 1/lambda (1).
26 // To produce a distribution with a different rate parameter
27 // callers can adjust the output using:
28 //
29 // sample = ExpFloat64() / desiredRateParameter
30 //
31 func (r *Rand) ExpFloat64() float64 {
32     for {
33         j := r.Uint32()
34         i := j & 0xFF
35         x := float64(j) * float64(we[i])
36         if j < ke[i] {
37             return x
38         }
39         if i == 0 {
40             return re - math.Log(r.Float64())
41         }
42         if fe[i]+float32(r.Float64())*(fe[i-1]-fe[i])
43             return x
44     }
45 }
```

```
45     }
46     panic("unreachable")
47 }
48
49 var ke = [256]uint32{
50     0xe290a139, 0x0, 0x9beadebc, 0xc377ac71, 0xd4ddb990,
51     0xde893fb8, 0xe4a8e87c, 0xe8dff16a, 0xebf2deab, 0xee
52     0xf0204efd, 0xf19bdb8e, 0xf2d458bb, 0xf3da104b, 0xf4
53     0xf577ad8a, 0xf61de83d, 0xf6afb784, 0xf730a573, 0xf7
54     0xf80a5bb6, 0xf867189d, 0xf8bb1b4f, 0xf9079062, 0xf9
55     0xf98d8c7d, 0xf9c8928a, 0xf9ff175b, 0xfa319996, 0xfa
56     0xfa8c3a62, 0xfab5084e, 0xfadb36c8, 0xfaff0410, 0xfb
57     0xfb404fb4, 0xfb5e2951, 0xfb7a59e9, 0xfb95038c, 0xfb
58     0xfbc638d8, 0xfbdcf892, 0xfbfb29a30, 0xfc0731df, 0xfc
59     0xfc2d8b02, 0xfc3f6c4d, 0xfc5083ac, 0xfc60ddd1, 0xfc
60     0xfc7f8810, 0xfc8decb4, 0xfc9bbd62, 0xfca9027c, 0xfc
61     0xfcc20864, 0xfccdd70a, 0xfcd935e3, 0xfce42ab0, 0xfc
62     0xfcf8eb3b, 0xfd02c0a0, 0xfd0c3f59, 0xfd156b7b, 0xfd
63     0xfd26daff, 0xfd2f2552, 0xfd372af7, 0xfd3e55e5, 0xfd
64     0xfd4dbc9e, 0xfd54cb85, 0xfd5ba2f2, 0xfd62451b, 0xfd
65     0xfd6ef1da, 0xfd750047, 0xfd7ae120, 0xfd809612, 0xfd
66     0xfd8b8285, 0xfd90bcf5, 0xfd95d15e, 0xfd9ac10b, 0xfd
67     0xfda43708, 0xfda8bf9e, 0xfdad2806, 0xfdb17141, 0xfd
68     0xfdb9a9fd, 0xfdbdb9b46, 0xfdc170f6, 0xfdc52bd8, 0xfd
69     0xfdcc542d, 0xfdcfc30b, 0xfdd319ef, 0xfdd6597a, 0xfd
70     0xfddc94e5, 0xfddf91e6, 0xfde279ce, 0xfde54d1f, 0xfd
71     0xfdeab7de, 0xfded5034, 0xfdefd5be, 0xfdf248e3, 0xfd
72     0xfdf6f984, 0xfdf937b6, 0xfdfb64f4, 0xfdfd818d, 0xfd
73     0xfe018a08, 0xfe03767a, 0xfe05536c, 0xfe07211c, 0xfe
74     0xfe0a8fab, 0xfe0c30fb, 0xfe0dc3ec, 0xfe0f48b1, 0xfe
75     0xfe122869, 0xfe1383b4, 0xfe14d17c, 0xfe1611e7, 0xfe
76     0xfe186b2a, 0xfe19843e, 0xfe1a9070, 0xfe1b8fd6, 0xfe
77     0xfe1d689b, 0xfe1e4220, 0xfe1f0f26, 0xfe1fcfbc, 0xfe
78     0xfe212bc3, 0xfe21c745, 0xfe225678, 0xfe22d95f, 0xfe
79     0xfe23ba4a, 0xfe241849, 0xfe2469f2, 0xfe24af3c, 0xfe
80     0xfe25148b, 0xfe253474, 0xfe2547c7, 0xfe254e70, 0xfe
81     0xfe25356a, 0xfe251586, 0xfe24e88f, 0xfe24ae64, 0xfe
82     0xfe2411df, 0xfe23af34, 0xfe233eb4, 0xfe22c02c, 0xfe
83     0xfe219838, 0xfe20ee58, 0xfe20358c, 0xfe1f6d92, 0xfe
84     0xfe1daef0, 0xfe1cb7ac, 0xfe1bb002, 0xfe1a9798, 0xfe
85     0xfe1832fd, 0xfe16e5fe, 0xfe15869d, 0xfe141464, 0xfe
86     0xfe10f565, 0xfe0f478c, 0xfe0d84b1, 0xfe0bac36, 0xfe
87     0xfe07b7b5, 0xfe059a40, 0xfe03644c, 0xfe011504, 0xfd
88     0xfdfc26e9, 0xfdf98629, 0xfdf6c83b, 0xfdf3ec01, 0xfd
89     0xfdedd3d1, 0xfdea953d, 0xfde7331e, 0xfde3abe9, 0xfd
90     0xfddc2791, 0xfdd826cd, 0xfdd3f9a8, 0xfdcf9dfc, 0xfd
91     0xfdc65198, 0xfdc15bb3, 0xfdbc2ce2, 0xfdb6c206, 0xfd
92     0xfdad2a63, 0xfda4f5fd, 0xfd9e7640, 0xfd97a67a, 0xfd
93     0xfd8901f2, 0xfd812182, 0xfd78d98e, 0xfd7022bb, 0xfd
94     0xfd5d4732, 0xfd530f9c, 0xfd48432b, 0xfd3cd59a, 0xfd
```

```

95         0xfd23dea4, 0xfd16349e, 0xfd07a7a3, 0xfcfc8219b, 0xfc
96         0xfcd5c220, 0xfcc2aadb, 0xfcfae1d5e, 0xfc97ed4e, 0xfc
97         0xfc65ccf3, 0xfc495762, 0xfc2a2fc8, 0xfc07ee19, 0xfb
98         0xfbb8051a, 0xfb890078, 0xfb5411a5, 0xfb180005, 0xfa
99         0xfa839276, 0xfa263b32, 0xf9b72d1c, 0xf930a1a2, 0xf8
100        0xf7b577d2, 0xf69c650c, 0xf51530f0, 0xf2cb0e3c, 0xee
101        0xe6da6ecf,
102    }
103    var we = [256]float32{
104        2.0249555e-09, 1.486674e-11, 2.4409617e-11, 3.196880
105        3.844677e-11, 4.4228204e-11, 4.9516443e-11, 5.443359
106        5.905944e-11, 6.344942e-11, 6.7643814e-11, 7.1672945
107        7.556032e-11, 7.932458e-11, 8.298079e-11, 8.654132e-
108        9.0016515e-11, 9.3415074e-11, 9.674443e-11, 1.000109
109        1.03220314e-10, 1.06377254e-10, 1.09486115e-10, 1.12
110        1.1557435e-10, 1.1856015e-10, 1.2151083e-10, 1.24428
111        1.2731648e-10, 1.3017575e-10, 1.3300853e-10, 1.35816
112        1.3860142e-10, 1.4136457e-10, 1.4410738e-10, 1.46831
113        1.4953687e-10, 1.5222583e-10, 1.54899e-10, 1.5755733
114        1.6020171e-10, 1.6283301e-10, 1.6545203e-10, 1.68059
115        1.7065617e-10, 1.732427e-10, 1.7581973e-10, 1.783878
116        1.8094774e-10, 1.8349985e-10, 1.8604476e-10, 1.88582
117        1.9111498e-10, 1.9364126e-10, 1.9616223e-10, 1.98678
118        2.0119004e-10, 2.0369768e-10, 2.0620168e-10, 2.08702
119        2.1120022e-10, 2.136955e-10, 2.1618855e-10, 2.186797
120        2.2116936e-10, 2.2365775e-10, 2.261452e-10, 2.286320
121        2.311185e-10, 2.3360494e-10, 2.360916e-10, 2.3857874
122        2.4106667e-10, 2.4355562e-10, 2.4604588e-10, 2.48537
123        2.5103128e-10, 2.5352695e-10, 2.560249e-10, 2.585254
124        2.6102867e-10, 2.6353494e-10, 2.6604446e-10, 2.68557
125        2.7107416e-10, 2.7359479e-10, 2.761196e-10, 2.786487
126        2.8118255e-10, 2.8372119e-10, 2.8626485e-10, 2.88813
127        2.9136826e-10, 2.939284e-10, 2.9649452e-10, 2.990667
128        3.016454e-10, 3.0423064e-10, 3.0682268e-10, 3.094217
129        3.1202813e-10, 3.1464195e-10, 3.1726352e-10, 3.19893
130        3.2253064e-10, 3.251767e-10, 3.2783135e-10, 3.304948
131        3.3316744e-10, 3.3584938e-10, 3.3854083e-10, 3.41242
132        3.4395342e-10, 3.46675e-10, 3.4940711e-10, 3.5215003
133        3.5490397e-10, 3.5766917e-10, 3.6044595e-10, 3.63234
134        3.660352e-10, 3.6884823e-10, 3.7167386e-10, 3.745124
135        3.773641e-10, 3.802293e-10, 3.8310827e-10, 3.860013e
136        3.8890866e-10, 3.918307e-10, 3.9476775e-10, 3.977200
137        4.0068804e-10, 4.0367196e-10, 4.0667217e-10, 4.09689
138        4.1272286e-10, 4.1577405e-10, 4.1884296e-10, 4.21929
139        4.250354e-10, 4.281597e-10, 4.313033e-10, 4.3446652e
140        4.3764986e-10, 4.408537e-10, 4.4407847e-10, 4.473246
141        4.5059267e-10, 4.5388301e-10, 4.571962e-10, 4.605326
142        4.6389292e-10, 4.6727755e-10, 4.70687e-10, 4.741219e
143        4.7758275e-10, 4.810702e-10, 4.845848e-10, 4.8812715

```

```
144 4.9169796e-10, 4.9529775e-10, 4.989273e-10, 5.025872
145 5.0627835e-10, 5.100013e-10, 5.1375687e-10, 5.175458
146 5.21369e-10, 5.2522725e-10, 5.2912136e-10, 5.330522e
147 5.370208e-10, 5.4102806e-10, 5.45075e-10, 5.491625e-
148 5.532918e-10, 5.5746385e-10, 5.616799e-10, 5.6594107
149 5.7024857e-10, 5.746037e-10, 5.7900773e-10, 5.834621
150 5.8796823e-10, 5.925276e-10, 5.971417e-10, 6.018122e
151 6.065408e-10, 6.113292e-10, 6.1617933e-10, 6.2109295
152 6.260722e-10, 6.3111916e-10, 6.3623595e-10, 6.414249
153 6.4668854e-10, 6.5202926e-10, 6.5744976e-10, 6.62952
154 6.6854156e-10, 6.742188e-10, 6.79988e-10, 6.858526e-
155 6.9181616e-10, 6.978826e-10, 7.04056e-10, 7.103407e-
156 7.167412e-10, 7.2326256e-10, 7.2990985e-10, 7.366886
157 7.4360473e-10, 7.5066453e-10, 7.5787476e-10, 7.65242
158 7.7277595e-10, 7.80483e-10, 7.883728e-10, 7.9645507e
159 8.047402e-10, 8.1323964e-10, 8.219657e-10, 8.309319e
160 8.401528e-10, 8.496445e-10, 8.594247e-10, 8.6951274e
161 8.799301e-10, 8.9070046e-10, 9.018503e-10, 9.134092e
162 9.254101e-10, 9.378904e-10, 9.508923e-10, 9.644638e-
163 9.786603e-10, 9.935448e-10, 1.0091913e-09, 1.025686e
164 1.0431306e-09, 1.0616465e-09, 1.08138e-09, 1.1025096
165 1.1252564e-09, 1.1498986e-09, 1.1767932e-09, 1.20640
166 1.2393786e-09, 1.276585e-09, 1.3193139e-09, 1.369543
167 1.4305498e-09, 1.508365e-09, 1.6160854e-09, 1.792124
168 }
169 var fe = [256]float32{
170 1, 0.9381437, 0.90046996, 0.87170434, 0.8477855, 0.8
171 0.8084217, 0.7915276, 0.77595687, 0.7614634, 0.74786
172 0.7350381, 0.72286767, 0.71127474, 0.70019263, 0.689
173 0.67935055, 0.6695063, 0.66000086, 0.65080583, 0.641
174 0.63325197, 0.6248527, 0.6166822, 0.60872537, 0.6009
175 0.5934009, 0.58601034, 0.5787874, 0.57172304, 0.5648
176 0.5580383, 0.5514034, 0.5448982, 0.5385169, 0.532253
177 0.5261042, 0.52006316, 0.5141264, 0.50828975, 0.5025
178 0.496902, 0.49134386, 0.485872, 0.48048335, 0.475175
179 0.46994483, 0.46478975, 0.45970762, 0.45469615, 0.44
180 0.44487688, 0.44006512, 0.43531612, 0.43062815, 0.42
181 0.42142874, 0.4169142, 0.41245446, 0.40804818, 0.403
182 0.3993907, 0.39513698, 0.39093173, 0.38677382, 0.382
183 0.37859577, 0.37457356, 0.37059465, 0.3666581, 0.362
184 0.35890847, 0.35509375, 0.351318, 0.3475805, 0.34388
185 0.34021714, 0.3365899, 0.33299807, 0.32944095, 0.325
186 0.3224285, 0.3189719, 0.31554767, 0.31215525, 0.3087
187 0.3054636, 0.3021634, 0.29889292, 0.2956517, 0.29243
188 0.28925523, 0.28609908, 0.28297043, 0.27986884, 0.27
189 0.2737453, 0.2707226, 0.2677254, 0.26475343, 0.26180
190 0.25888354, 0.25598502, 0.2531103, 0.25025907, 0.247
191 0.24462597, 0.24184346, 0.23908329, 0.23634516, 0.23
192 0.23093392, 0.2282603, 0.22560766, 0.22297576, 0.220
```

```
193      0.21777324, 0.21520215, 0.21265087, 0.21011916, 0.20
194      0.20511365, 0.20263945, 0.20018397, 0.19774707, 0.19
195      0.19292815, 0.19054577, 0.1881812, 0.18583426, 0.183
196      0.1811926, 0.17889754, 0.17661946, 0.17435817, 0.172
197      0.1698854, 0.16767362, 0.16547804, 0.16329853, 0.161
198      0.15898713, 0.15685499, 0.15473837, 0.15263714, 0.15
199      0.14848037, 0.14642459, 0.14438373, 0.14235765, 0.14
200      0.13834943, 0.13636707, 0.13439907, 0.13244532, 0.13
201      0.1285802, 0.12666863, 0.12477092, 0.12288698, 0.121
202      0.119160056, 0.1173169, 0.115487166, 0.11367077, 0.1
203      0.11007768, 0.10830083, 0.10653701, 0.10478614, 0.10
204      0.101323, 0.09961058, 0.09791085, 0.09622374, 0.0945
205      0.09288713, 0.091237515, 0.08960028, 0.087975375, 0.
206      0.08476233, 0.083174095, 0.081597984, 0.08003395, 0.
207      0.076941945, 0.07541389, 0.07389775, 0.072393484, 0.
208      0.069420435, 0.06795159, 0.066494495, 0.06504912, 0.
209      0.062193416, 0.060783047, 0.059384305, 0.057997175,
210      0.05662164, 0.05525769, 0.053905312, 0.052564494, 0.
211      0.049917534, 0.048611384, 0.047316793, 0.046033762,
212      0.043502413, 0.042254124, 0.041017443, 0.039792392,
213      0.038578995, 0.037377283, 0.036187284, 0.035009038,
214      0.033842582, 0.032687962, 0.031545233, 0.030414443,
215      0.02818895, 0.027094385, 0.026012046, 0.024942026, 0
216      0.022839336, 0.021806888, 0.020787204, 0.019780423,
217      0.0178062, 0.016839107, 0.015885621, 0.014945968, 0.
218      0.013109165, 0.012212592, 0.011331013, 0.01046481, 0
219      0.008780315, 0.007963077, 0.0071633533, 0.006381906,
220      0.0056196423, 0.0048776558, 0.004157295, 0.003460264
221      0.0027887989, 0.0021459677, 0.0015362998, 0.00096726
222      0.00045413437,
223 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/rand/normal.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package rand
6
7 import (
8     "math"
9 )
10
11 /*
12  * Normal distribution
13  *
14  * See "The Ziggurat Method for Generating Random Variables"
15  * (Marsaglia & Tsang, 2000)
16  * http://www.jstatsoft.org/v05/i08/paper [pdf]
17  */
18
19 const (
20     rn = 3.442619855899
21 )
22
23 func absInt32(i int32) uint32 {
24     if i < 0 {
25         return uint32(-i)
26     }
27     return uint32(i)
28 }
29
30 // NormFloat64 returns a normally distributed float64 in the
31 // [-math.MaxFloat64, +math.MaxFloat64] with
32 // standard normal distribution (mean = 0, stddev = 1).
33 // To produce a different normal distribution, callers can
34 // adjust the output using:
35 //
36 // sample = NormFloat64() * desiredStdDev + desiredMean
37 //
38 func (r *Rand) NormFloat64() float64 {
39     for {
40         j := int32(r.Uint32()) // Possibly negative
41         i := j & 0x7F
```

```

42         x := float64(j) * float64(wn[i])
43         if absInt32(j) < kn[i] {
44             // This case should be hit better th
45             return x
46         }
47
48         if i == 0 {
49             // This extra work is only required
50             for {
51                 x = -math.Log(r.Float64()) *
52                 y := -math.Log(r.Float64())
53                 if y+y >= x*x {
54                     break
55                 }
56             }
57             if j > 0 {
58                 return rn + x
59             }
60             return -rn - x
61         }
62         if fn[i]+float32(r.Float64())*(fn[i-1]-fn[i]
63         return x
64     }
65 }
66     panic("unreachable")
67 }
68
69 var kn = [128]uint32{
70     0x76ad2212, 0x0, 0x600f1b53, 0x6ce447a6, 0x725b46a2,
71     0x7560051d, 0x774921eb, 0x789a25bd, 0x799045c3, 0x7a
72     0x7adf629f, 0x7b5682a6, 0x7bb8a8c6, 0x7c0ae722, 0x7c
73     0x7c8cec5b, 0x7cc12cd6, 0x7ceefed2, 0x7d177e0b, 0x7d
74     0x7d5bce6c, 0x7d78dd64, 0x7d932886, 0x7dab0e57, 0x7d
75     0x7dd4d688, 0x7de73185, 0x7df81cea, 0x7e07c0a3, 0x7e
76     0x7e23b587, 0x7e303dfd, 0x7e3beec2, 0x7e46db77, 0x7e
77     0x7e5aabb3, 0x7e63abf7, 0x7e6c222c, 0x7e741906, 0x7e
78     0x7e82adfa, 0x7e895c63, 0x7e8fac4b, 0x7e95a3fb, 0x7e
79     0x7ea0a0ef, 0x7ea5b00d, 0x7eaa7ac3, 0x7eaf04f3, 0x7e
80     0x7eb765a5, 0x7ebb4259, 0x7ebeeafd, 0x7ec2620a, 0x7e
81     0x7ec8c441, 0x7ecbb365, 0x7ece78ed, 0x7ed11671, 0x7e
82     0x7ed5df12, 0x7ed80cb4, 0x7eda175c, 0x7edc0005, 0x7e
83     0x7edf6ebf, 0x7ee0f647, 0x7ee25ebe, 0x7ee3a8a9, 0x7e
84     0x7ee5e276, 0x7ee6d2f5, 0x7ee7a620, 0x7ee85c10, 0x7e
85     0x7ee97047, 0x7ee9ce59, 0x7eea0eca, 0x7eea3147, 0x7e
86     0x7eea1aab, 0x7ee9e071, 0x7ee98602, 0x7ee90a88, 0x7e
87     0x7ee7ac6a, 0x7ee6c769, 0x7ee5bc9c, 0x7ee48a67, 0x7e
88     0x7ee1a857, 0x7edff42f, 0x7ede0ffa, 0x7edbf8d9, 0x7e
89     0x7ed7248d, 0x7ed45fae, 0x7ed1585c, 0x7ece095f, 0x7e
90     0x7ec67be2, 0x7ec22eee, 0x7ebd7d1a, 0x7eb85c35, 0x7e
91     0x7eac9c20, 0x7ea5df27, 0x7e9e769f, 0x7e964c16, 0x7e

```

```

92         0x7e834033, 0x7e781728, 0x7e6b9933, 0x7e5d8a1a, 0x7e
93         0x7e3b737a, 0x7e268c2f, 0x7e0e3ff5, 0x7df1aa5d, 0x7d
94         0x7da61a1e, 0x7d72a0fb, 0x7d30e097, 0x7cd9b4ab, 0x7c
95         0x7ba90bdc, 0x7a722176, 0x77d664e5,
96     }
97     var wn = [128]float32{
98         1.7290405e-09, 1.2680929e-10, 1.6897518e-10, 1.98626
99         2.2232431e-10, 2.4244937e-10, 2.601613e-10, 2.761198
100        2.9073963e-10, 3.042997e-10, 3.1699796e-10, 3.289802
101        3.4035738e-10, 3.5121603e-10, 3.616251e-10, 3.716405
102        3.8130857e-10, 3.9066758e-10, 3.9975012e-10, 4.08584
103        4.1719309e-10, 4.2559822e-10, 4.338176e-10, 4.418672
104        4.497613e-10, 4.5751258e-10, 4.651324e-10, 4.7263105
105        4.8001775e-10, 4.87301e-10, 4.944885e-10, 5.015873e-
106        5.0860405e-10, 5.155446e-10, 5.2241467e-10, 5.292193
107        5.359635e-10, 5.426517e-10, 5.4928817e-10, 5.5587696
108        5.624219e-10, 5.6892646e-10, 5.753941e-10, 5.818282e
109        5.882317e-10, 5.946077e-10, 6.00959e-10, 6.072884e-1
110        6.135985e-10, 6.19892e-10, 6.2617134e-10, 6.3243905e
111        6.386974e-10, 6.449488e-10, 6.511956e-10, 6.5744005e
112        6.6368433e-10, 6.699307e-10, 6.7618144e-10, 6.824387
113        6.8870465e-10, 6.949815e-10, 7.012715e-10, 7.075768e
114        7.1389966e-10, 7.202424e-10, 7.266073e-10, 7.329966e
115        7.394128e-10, 7.4585826e-10, 7.5233547e-10, 7.58847e
116        7.653954e-10, 7.719835e-10, 7.7861395e-10, 7.852897e
117        7.920138e-10, 7.987892e-10, 8.0561924e-10, 8.125073e
118        8.194569e-10, 8.2647167e-10, 8.3355556e-10, 8.407127
119        8.479473e-10, 8.55264e-10, 8.6266755e-10, 8.7016316e
120        8.777562e-10, 8.8545243e-10, 8.932582e-10, 9.0117996
121        9.09225e-10, 9.174008e-10, 9.2571584e-10, 9.341788e-
122        9.427997e-10, 9.515889e-10, 9.605579e-10, 9.697193e-
123        9.790869e-10, 9.88676e-10, 9.985036e-10, 1.0085882e-
124        1.0189509e-09, 1.0296151e-09, 1.0406069e-09, 1.05195
125        1.063698e-09, 1.0758702e-09, 1.0885183e-09, 1.101694
126        1.1154611e-09, 1.1298902e-09, 1.1450696e-09, 1.16110
127        1.1781276e-09, 1.1962995e-09, 1.2158287e-09, 1.23698
128        1.2601323e-09, 1.2857697e-09, 1.3146202e-09, 1.34778
129        1.3870636e-09, 1.4357403e-09, 1.5008659e-09, 1.60309
130     }
131     var fn = [128]float32{
132         1, 0.9635997, 0.9362827, 0.9130436, 0.89228165, 0.87
133         0.8555006, 0.8387836, 0.8229072, 0.8077383, 0.793177
134         0.7791461, 0.7655842, 0.7524416, 0.73967725, 0.72725
135         0.7151515, 0.7033361, 0.69178915, 0.68049186, 0.6694
136         0.658582, 0.6479418, 0.63749546, 0.6272325, 0.617143
137         0.6072195, 0.5974532, 0.58783704, 0.5783647, 0.56903
138         0.5598274, 0.5507518, 0.54179835, 0.5329627, 0.52424
139         0.5156282, 0.50712204, 0.49871865, 0.49041483, 0.482
140         0.4740943, 0.46607214, 0.4581387, 0.45029163, 0.4425

```

```
141      0.43484783, 0.427247, 0.41972435, 0.41227803, 0.4049
142      0.39760786, 0.3903808, 0.3832238, 0.37613547, 0.3691
143      0.3621595, 0.35526937, 0.34844297, 0.34167916, 0.334
144      0.3283351, 0.3217529, 0.3152294, 0.30876362, 0.30235
145      0.29600215, 0.28970486, 0.2834622, 0.2772735, 0.2711
146      0.2650553, 0.25902456, 0.2530453, 0.24711695, 0.2412
147      0.23541094, 0.22963232, 0.2239027, 0.21822165, 0.212
148      0.20700371, 0.20146611, 0.19597565, 0.19053204, 0.18
149      0.17978427, 0.17447963, 0.1692209, 0.16400786, 0.158
150      0.15371831, 0.14864157, 0.14361008, 0.13862377, 0.13
151      0.12878671, 0.12393598, 0.119130544, 0.11437051, 0.1
152      0.104987256, 0.10036444, 0.095787846, 0.0912578, 0.0
153      0.0823389, 0.077950984, 0.073611505, 0.06932112, 0.0
154      0.06089077, 0.056752663, 0.0526674, 0.048636295, 0.0
155      0.040742867, 0.03688439, 0.033087887, 0.029356318,
156      0.025693292, 0.022103304, 0.018592102, 0.015167298,
157      0.011839478, 0.008624485, 0.005548995, 0.0026696292,
158 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/math/rand/rand.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package rand implements pseudo-random number generators.
6 package rand
7
8 import "sync"
9
10 // A Source represents a source of uniformly-distributed
11 // pseudo-random int64 values in the range [0, 1<<63).
12 type Source interface {
13     Int63() int64
14     Seed(seed int64)
15 }
16
17 // NewSource returns a new pseudo-random Source seeded with
18 func NewSource(seed int64) Source {
19     var rng rngSource
20     rng.Seed(seed)
21     return &rng
22 }
23
24 // A Rand is a source of random numbers.
25 type Rand struct {
26     src Source
27 }
28
29 // New returns a new Rand that uses random values from src
30 // to generate other random values.
31 func New(src Source) *Rand { return &Rand{src} }
32
33 // Seed uses the provided seed value to initialize the gener
34 func (r *Rand) Seed(seed int64) { r.src.Seed(seed) }
35
36 // Int63 returns a non-negative pseudo-random 63-bit integer
37 func (r *Rand) Int63() int64 { return r.src.Int63() }
38
39 // Uint32 returns a pseudo-random 32-bit value as a uint32.
40 func (r *Rand) Uint32() uint32 { return uint32(r.Int63() >>
41
```

```

42 // Int31 returns a non-negative pseudo-random 31-bit integer
43 func (r *Rand) Int31() int32 { return int32(r.Int63() >> 32)
44
45 // Int returns a non-negative pseudo-random int.
46 func (r *Rand) Int() int {
47     u := uint(r.Int63())
48     return int(u << 1 >> 1) // clear sign bit if int ==
49 }
50
51 // Int63n returns, as an int64, a non-negative pseudo-random
52 // It panics if n <= 0.
53 func (r *Rand) Int63n(n int64) int64 {
54     if n <= 0 {
55         panic("invalid argument to Int63n")
56     }
57     max := int64((1 << 63) - 1 - (1<<63)%uint64(n))
58     v := r.Int63()
59     for v > max {
60         v = r.Int63()
61     }
62     return v % n
63 }
64
65 // Int31n returns, as an int32, a non-negative pseudo-random
66 // It panics if n <= 0.
67 func (r *Rand) Int31n(n int32) int32 {
68     if n <= 0 {
69         panic("invalid argument to Int31n")
70     }
71     max := int32((1 << 31) - 1 - (1<<31)%uint32(n))
72     v := r.Int31()
73     for v > max {
74         v = r.Int31()
75     }
76     return v % n
77 }
78
79 // Intn returns, as an int, a non-negative pseudo-random num
80 // It panics if n <= 0.
81 func (r *Rand) Intn(n int) int {
82     if n <= 0 {
83         panic("invalid argument to Intn")
84     }
85     if n <= 1<<31-1 {
86         return int(r.Int31n(int32(n)))
87     }
88     return int(r.Int63n(int64(n)))
89 }
90
91 // Float64 returns, as a float64, a pseudo-random number in

```

```

92 func (r *Rand) Float64() float64 { return float64(r.Int63())
93
94 // Float32 returns, as a float32, a pseudo-random number in
95 func (r *Rand) Float32() float32 { return float32(r.Float64(
96
97 // Perm returns, as a slice of n ints, a pseudo-random permu
98 func (r *Rand) Perm(n int) []int {
99     m := make([]int, n)
100    for i := 0; i < n; i++ {
101        m[i] = i
102    }
103    for i := 0; i < n; i++ {
104        j := r.Intn(i + 1)
105        m[i], m[j] = m[j], m[i]
106    }
107    return m
108 }
109
110 /*
111  * Top-level convenience functions
112  */
113
114 var globalRand = New(&lockedSource{src: NewSource(1)})
115
116 // Seed uses the provided seed value to initialize the gener
117 // deterministic state. If Seed is not called, the generator
118 // if seeded by Seed(1).
119 func Seed(seed int64) { globalRand.Seed(seed) }
120
121 // Int63 returns a non-negative pseudo-random 63-bit integer
122 func Int63() int64 { return globalRand.Int63() }
123
124 // Uint32 returns a pseudo-random 32-bit value as a uint32.
125 func Uint32() uint32 { return globalRand.Uint32() }
126
127 // Int31 returns a non-negative pseudo-random 31-bit integer
128 func Int31() int32 { return globalRand.Int31() }
129
130 // Int returns a non-negative pseudo-random int.
131 func Int() int { return globalRand.Int() }
132
133 // Int63n returns, as an int64, a non-negative pseudo-random
134 // It panics if n <= 0.
135 func Int63n(n int64) int64 { return globalRand.Int63n(n) }
136
137 // Int31n returns, as an int32, a non-negative pseudo-random
138 // It panics if n <= 0.
139 func Int31n(n int32) int32 { return globalRand.Int31n(n) }
140

```

```

141 // Intn returns, as an int, a non-negative pseudo-random num
142 // It panics if n <= 0.
143 func Intn(n int) int { return globalRand.Intn(n) }
144
145 // Float64 returns, as a float64, a pseudo-random number in
146 func Float64() float64 { return globalRand.Float64() }
147
148 // Float32 returns, as a float32, a pseudo-random number in
149 func Float32() float32 { return globalRand.Float32() }
150
151 // Perm returns, as a slice of n ints, a pseudo-random permu
152 func Perm(n int) []int { return globalRand.Perm(n) }
153
154 // NormFloat64 returns a normally distributed float64 in the
155 // [-math.MaxFloat64, +math.MaxFloat64] with
156 // standard normal distribution (mean = 0, stddev = 1).
157 // To produce a different normal distribution, callers can
158 // adjust the output using:
159 //
160 // sample = NormFloat64() * desiredStdDev + desiredMean
161 //
162 func NormFloat64() float64 { return globalRand.NormFloat64() }
163
164 // ExpFloat64 returns an exponentially distributed float64 i
165 // (0, +math.MaxFloat64] with an exponential distribution wh
166 // (lambda) is 1 and whose mean is 1/lambda (1).
167 // To produce a distribution with a different rate parameter
168 // callers can adjust the output using:
169 //
170 // sample = ExpFloat64() / desiredRateParameter
171 //
172 func ExpFloat64() float64 { return globalRand.ExpFloat64() }
173
174 type lockedSource struct {
175     lk sync.Mutex
176     src Source
177 }
178
179 func (r *lockedSource) Int63() (n int64) {
180     r.lk.Lock()
181     n = r.src.Int63()
182     r.lk.Unlock()
183     return
184 }
185
186 func (r *lockedSource) Seed(seed int64) {
187     r.lk.Lock()
188     r.src.Seed(seed)
189     r.lk.Unlock()

```

190 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/rand/rng.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package rand
6
7 /*
8  * Uniform distribution
9  *
10 * algorithm by
11 * DP Mitchell and JA Reeds
12 */
13
14 const (
15     _LEN  = 607
16     _TAP  = 273
17     _MAX  = 1 << 63
18     _MASK = _MAX - 1
19     _A    = 48271
20     _M    = (1 << 31) - 1
21     _Q    = 44488
22     _R    = 3399
23 )
24
25 var (
26     // cooked random numbers
27     // the state of the rng
28     // after 780e10 iterations
29     rng_cooked [_LEN]int64 = [...]int64{
30         5041579894721019882, 4646389086726545243, 13
31         2875692520355975054, 9033628115061424579, 71
32         7937252194349799378, 5307299880338848416, 82
33         4593015457530856296, 8140875735541888011, 33
34         1727074043483619500, 113108499721038619, 456
35         2387618771259064424, 2716131344356686112, 65
36         7684323884043752161, 257867835996031390, 659
37         2789386447340118284, 1065192797246149621, 33
38         7465081662728599889, 1014950805555097187, 44
39         2418672789110888383, 5796562887576294778, 44
40         4523597184512354423, 10530508058128498, 8633
41         8660405965245884302, 10162832508971942, 6540
42         6240911277345944669, 831864355460801054, 800
43         2202309800992166967, 9161020366945053561, 40
44         457351505131524928, 342195045928179354, 2847
```

45 4368649989588021065, 887231587095185257, 556
46 5616972787034086048, 8471809303394836566, 16
47 4244156215201778923, 7848217333783577387, 56
48 9029726508369077193, 3243583134664087292, 43
49 6446940406810434101, 1679342092332374735, 60
50 7640877852514293609, 5881353426285907985, 81
51 2725470216277009086, 4980675660146853729, 52
52 6326442804750084282, 1495812843684243920, 70
53 6756929275356942261, 4706794511633873654, 78
54 33650829478596156, 1328918435751322643, 7297
55 2238025036817854944, 5147159997473910359, 89
56 6097729358393448602, 1731725986304753684, 41
57 8477511620686074402, 5803876044675762232, 84
58 4715837297103951910, 7566171971264485114, 50
59 842110666775871513, 572156825025677802, 1791
60 3778721850472236509, 2352769483186201278, 12
61 5781809037144163536, 2733958794029492513, 50
62 4234737173186232084, 5027558287275472836, 46
63 5907508150730407386, 784756255473944452, 972
64 5158420642969283891, 9048531678558643225, 24
65 3940796514530962282, 3341174631045206375, 30
66 5832080132947175283, 7890064875145919662, 81
67 1464597243840211302, 4641648007187991873, 35
68 6657089965014657519, 5220884358887979358, 17
69 1147977171614181568, 5066037465548252321, 25
70 3350107529868390359, 6116438694366558490, 21
71 2469478054175558874, 7368243281019965984, 37
72 2257095756513439648, 7217693971077460129, 90
73 5637660345400869599, 3955544945427965183, 80
74 6621926588513568059, 1373361136802681441, 65
75 9202058512774729859, 1954818376891585542, 66
76 3901867355218954373, 7046310742295574065, 68
77 8850422670118399721, 3631909142291992901, 51
78 4763258931815816403, 6280052734341785344, 42
79 6545300466022085465, 4562580375758598164, 54
80 553004618757816492, 6895160632757959823, 823
81 8550891222387991669, 5535668688139305547, 24
82 8159640039107728799, 6157493831600770366, 76
83 3681878764086140361, 3289686137190109749, 65
84 4079788377417136100, 8090302575944624335, 29
85 3009039260312620700, 8430027460082534031, 40
86 4707864159563588614, 5640248530963493951, 59
87 5503847578771918426, 3941971367175193882, 81
88 7062410411742090847, 741381002980207668, 602
89 6251390334426228834, 1368930247903518833, 88
90 2462145737463489636, 404828418920299174, 415
91 5464715384600071357, 592710404378763017, 676
92 5820343663801914208, 385298524683789911, 522
93 7150122561309371392, 368107899140673753, 311
94 4782583894627718279, 6718292300699989587, 83

95	4654329375432538231,	8930667561363381602,	53
96	7725442901813263321,	9186225467561587250,	40
97	2530936820058611833,	1636551876240043639,	55
98	2061642381019690829,	1279580266495294036,	91
99	5007630032676973346,	2153168792952589781,	67
100	3433922409283786309,	2285479922797300912,	31
101	5418791419666136509,	7163298419643543757,	48
102	1684034065251686769,	4429514767357295841,	33
103	7177515271653460134,	4589042248470800257,	76
104	246994305896273627,	866417324803099287,	6473
105	2058427839513754051,	5133784708526867938,	87
106	8585842181454472135,	6137678347805511274,	20
107	5999657892458244504,	4358391411789012426,	32
108	4843721905315627004,	6010651222149276415,	53
109	1044646352569048800,	9106614159853161675,	82
110	2681532557646850893,	3681559472488511871,	53
111	2658708232916537604,	1163313865052186287,	58
112	4423673246306544414,	1620799783996955743,	22
113	4287360518296753003,	4590000184845883843,	55
114	478991688350776035,	8746140185685648781,	228
115	3019253992034194581,	3152601605678500003,	43
116	4916432985369275664,	663574931734554391,	342
117	1999319134044418520,	3328689518636282723,	25
118	3092343956317362483,	3662252519007064108,	97
119	1708913533482282562,	6917817162668868494,	32
120	8739788839543624613,	2488075924621352812,	46
121	2997203966153298104,	1282559373026354493,	24
122	628141331766346752,	4571950817186770476,	147
123	6091219417754424743,	7834161864828164065,	71
124	4442653864240571734,	8903482404847331368,	62
125	504404948065709118,	7275215526217113061,	101
126	2623071828615234808,	5157313728073836108,	37
127	2467284396349269342,	5256026990536851868,	78
128	1202087339038317498,	2113514992440715978,	75
129	5145623771477493805,	8225140880134972332,	27
130	4332154495710163773,	7137789594094346916,	69
131	655440045726677499,	59934747298466858,	61249
132	2332206071942466437,	1701056712286369627,	31
133	2460521277767576533,	197309393502684135,	643
134	4350769010207485119,	4754652089410667672,	20
135	4287946071480840813,	8362686366770308971,	64
136	7554353525834302244,	4450022655153542367,	16
137	4626575813550328320,	2692222020597705149,	24
138	7916882295460730264,	884817090297530579,	532
139	4955070238059373407,	4918537275422674302,	30
140	2470346235617803020,	8928702772696731736,	78
141	7900176660600710914,	2140571127916226672,	24
142	4186670094382025798,	1883939007446035042,	88
143	4065968871360089196,	6953124200385847784,	13

```

144     3656125660947993209, 3966759634633167020, 31
145     4565385105440252958, 1979884289539493806, 23
146     8464961209802336085, 2843963751609577687, 30
147     4459239494808162889, 402587895800087237, 805
148     1042662272908816815, 5557303057122568958, 26
149     5806352215355387087, 7117771003473903623, 59
150     8833658097025758785, 5970273481425315300, 56
151     1598828206250873866, 5206393647403558110, 62
152     8469693267274066490, 125672920241807416, 531
153     8736848295048751716, 4488039774992061878, 59
154     7414942793393574290, 7990420780896957397, 43
155     2740722765288122703, 5743100009702758344, 59
156     5242208035432907801, 701338899890987198, 760
157     6651322707055512866, 2635195723621160615, 51
158     1567242097116389047, 8172389260191636581, 63
159     2743190902890262681, 1906367633221323427, 60
160     2241128460406315459, 895504896216695588, 309
161     9079887171656594975, 8839289181930711403, 57
162     1838220598389033063, 3801620336801580414, 88
163     7899055018877642622, 5421679761463003041, 55
164     8735487530905098534, 1760527091573692978, 71
165     4969531564923704323, 3394475942196872480, 64
166     3273651282831730598, 6797106199797138596, 30
167     6036575856065626233, 740416251634527158, 708
168     399922722363687927, 294902314447253185, 7844
169     6192655680808675579, 411604686384710388, 902
170     4615674634722404292, 539897290441580544, 209
171     1907224908052289603, 7381039757301768559, 61
172     8629571604380892756, 5280433031239081479, 71
173     7169176924412769570, 7942066497793203302, 13
174     3625338785743880657, 6477479539006708521, 89
175     1326024180520890843, 7537449876596048829, 54
176     6346751753565857109, 241159987320630307, 309
177     2902794662273147216, 7208698530190629697, 72
178     4133292154170828382, 2918308698224194548, 15
179     4922828954023452664, 2879211533496425641, 58
180     7329020396871624740, 8915471717014488588, 29
181     8382142935188824023, 9103922860780351547, 41
182     }
183 )
184
185 type rngSource struct {
186     tap int // index into vec
187     feed int // index into vec
188     vec [_LEN]int64 // current feedback register
189 }
190
191 // seed rng x[n+1] = 48271 * x[n] mod (2**31 - 1)
192 func seedrand(x int32) int32 {

```

```

193         hi := x / _Q
194         lo := x % _Q
195         x = _A*lo - _R*hi
196         if x < 0 {
197             x += _M
198         }
199         return x
200     }
201
202 // Seed uses the provided seed value to initialize the gener
203 func (rng *rngSource) Seed(seed int64) {
204     rng.tap = 0
205     rng.feed = _LEN - _TAP
206
207     seed = seed % _M
208     if seed < 0 {
209         seed += _M
210     }
211     if seed == 0 {
212         seed = 89482311
213     }
214
215     x := int32(seed)
216     for i := -20; i < _LEN; i++ {
217         x = seedrand(x)
218         if i >= 0 {
219             var u int64
220             u = int64(x) << 40
221             x = seedrand(x)
222             u ^= int64(x) << 20
223             x = seedrand(x)
224             u ^= int64(x)
225             u ^= rng_cooked[i]
226             rng.vec[i] = u & _MASK
227         }
228     }
229 }
230
231 // Int63 returns a non-negative pseudo-random 63-bit integer
232 func (rng *rngSource) Int63() int64 {
233     rng.tap--
234     if rng.tap < 0 {
235         rng.tap += _LEN
236     }
237
238     rng.feed--
239     if rng.feed < 0 {
240         rng.feed += _LEN
241     }
242

```

```
243         x := (rng.vec[rng.feed] + rng.vec[rng.tap]) & _MASK
244         rng.vec[rng.feed] = x
245         return x
246     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/math/rand/zipf.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // W.Hormann, G.Derflinger:
6 // "Rejection-Inversion to Generate Variates
7 // from Monotone Discrete Distributions"
8 // http://eeyore.wu-wien.ac.at/papers/96-04-04.wh-der.ps.gz
9
10 package rand
11
12 import "math"
13
14 // A Zipf generates Zipf distributed variates.
15 type Zipf struct {
16     r          *Rand
17     imax       float64
18     v          float64
19     q          float64
20     s          float64
21     oneminusQ float64
22     oneminusQinv float64
23     hxm        float64
24     hx0minusHxm float64
25 }
26
27 func (z *Zipf) h(x float64) float64 {
28     return math.Exp(z.oneminusQ*math.Log(z.v+x)) * z.one
29 }
30
31 func (z *Zipf) hinv(x float64) float64 {
32     return math.Exp(z.oneminusQinv*math.Log(z.oneminusQ+x))
33 }
34
35 // NewZipf returns a Zipf generating variates p(k) on [0, imax]
36 // proportional to (v+k)**(-s) where s>1 and k>=0, and v>=1.
37 //
38 func NewZipf(r *Rand, s float64, v float64, imax uint64) *Zipf {
39     z := new(Zipf)
40     if s <= 1.0 || v < 1 {
41         return nil
42     }
43     z.r = r
44     z.imax = float64(imax)
```

```

45         z.v = v
46         z.q = s
47         z.oneminusQ = 1.0 - z.q
48         z.oneminusQinv = 1.0 / z.oneminusQ
49         z.hxm = z.h(z.imax + 0.5)
50         z.hx0minusHxm = z.h(0.5) - math.Exp(math.Log(z.v)*(-
51         z.s = 1 - z.hinv(z.h(1.5)-math.Exp(-z.q*math.Log(z.v)
52         return z
53     }
54
55     // Uint64 returns a value drawn from the Zipf distributed de
56     // by the Zipf object.
57     func (z *Zipf) Uint64() uint64 {
58         k := 0.0
59
60         for {
61             r := z.r.Float64() // r on [0,1]
62             ur := z.hxm + r*z.hx0minusHxm
63             x := z.hinv(ur)
64             k = math.Floor(x + 0.5)
65             if k-x <= z.s {
66                 break
67             }
68             if ur >= z.h(k+0.5)-math.Exp(-math.Log(k+z.v
69                 break
70         }
71     }
72     return uint64(k)
73 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/mime/grammar.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package mime
6
7 import (
8     "strings"
9 )
10
11 // isTSpecial returns true if rune is in 'tspecials' as defi
12 // 1521 and RFC 2045.
13 func isTSpecial(r rune) bool {
14     return strings.IndexRune(`()<>@,;:\"/[]?=\`, r) != -1
15 }
16
17 // isTokenChar returns true if rune is in 'token' as defined
18 // 1521 and RFC 2045.
19 func isTokenChar(r rune) bool {
20     // token := 1*<any (US-ASCII) CHAR except SPACE, CTL
21     //           or tspecials>
22     return r > 0x20 && r < 0x7f && !isTSpecial(r)
23 }
24
25 // isToken returns true if s is a 'token' as as defined by R
26 // and RFC 2045.
27 func isToken(s string) bool {
28     if s == "" {
29         return false
30     }
31     return strings.IndexFunc(s, isNotTokenChar) < 0
32 }
33
34 // isQText returns true if rune is in 'qtext' as defined by
35 func isQText(r int) bool {
36     // CHAR           = <any ASCII character>           ; (
37     // qtext          = <any CHAR excepting <">,       ; =>
38     //                "\" & CR, and including
39     //                linear-white-space>
40     switch r {
41     case '\'', '\\', '\r':
42         return false
43     }
44     return r < 0x80
```

45 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/mime/mediatype.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package mime
6
7 import (
8     "bytes"
9     "errors"
10    "fmt"
11    "strings"
12    "unicode"
13 )
14
15 // FormatMediaType serializes mediatype t and the parameters
16 // param as a media type conforming to RFC 2045 and RFC 2616
17 // The type and parameter names are written in lower-case.
18 // When any of the arguments result in a standard violation
19 // FormatMediaType returns the empty string.
20 func FormatMediaType(t string, param map[string]string) string {
21     slash := strings.Index(t, "/")
22     if slash == -1 {
23         return ""
24     }
25     major, sub := t[:slash], t[slash+1:]
26     if !isToken(major) || !isToken(sub) {
27         return ""
28     }
29     var b bytes.Buffer
30     b.WriteString(strings.ToLower(major))
31     b.WriteByte('/')
32     b.WriteString(strings.ToLower(sub))
33
34     for attribute, value := range param {
35         b.WriteByte(';')
36         b.WriteByte(' ')
37         if !isToken(attribute) {
38             return ""
39         }
40         b.WriteString(strings.ToLower(attribute))
41         b.WriteByte('=')
```

```

42         if isToken(value) {
43             b.WriteString(value)
44             continue
45         }
46
47         b.WriteByte('"')
48         offset := 0
49         for index, character := range value {
50             if character == '"' || character ==
51                 b.WriteString(value[offset:i
52                 offset = index
53                 b.WriteByte('\\')
54             }
55             if character&0x80 != 0 {
56                 return ""
57             }
58         }
59         b.WriteString(value[offset:])
60         b.WriteByte('"')
61     }
62     return b.String()
63 }
64
65 func checkMediaTypeDisposition(s string) error {
66     typ, rest := consumeToken(s)
67     if typ == "" {
68         return errors.New("mime: no media type")
69     }
70     if rest == "" {
71         return nil
72     }
73     if !strings.HasPrefix(rest, "/") {
74         return errors.New("mime: expected slash after
75     }
76     subtype, rest := consumeToken(rest[1:])
77     if subtype == "" {
78         return errors.New("mime: expected token after
79     }
80     if rest != "" {
81         return errors.New("mime: unexpected content
82     }
83     return nil
84 }
85
86 // ParseMediaType parses a media type value and any optional
87 // parameters, per RFC 1521. Media types are the values in
88 // Content-Type and Content-Disposition headers (RFC 2183).
89 // On success, ParseMediaType returns the media type converted
90 // to lowercase and trimmed of white space and a non-nil map
91 // The returned map, params, maps from the lowercase

```

```

92 // attribute to the attribute value with its case preserved.
93 func ParseMediaType(v string) (mediatype string, params map[
94     i := strings.Index(v, ";")
95     if i == -1 {
96         i = len(v)
97     }
98     mediatype = strings.TrimSpace(strings.ToLower(v[0:i])
99
100     err = checkMediaTypeDisposition(mediatype)
101     if err != nil {
102         return "", nil, err
103     }
104
105     params = make(map[string]string)
106
107     // Map of base parameter name -> parameter name -> v
108     // for parameters containing a '*' character.
109     // Lazily initialized.
110     var continuation map[string]map[string]string
111
112     v = v[i:]
113     for len(v) > 0 {
114         v = strings.TrimLeftFunc(v, unicode.IsSpace)
115         if len(v) == 0 {
116             break
117         }
118         key, value, rest := consumeMediaParam(v)
119         if key == "" {
120             if strings.TrimSpace(rest) == ";" {
121                 // Ignore trailing semicolon
122                 // Not an error.
123                 return
124             }
125             // Parse error.
126             return "", nil, errors.New("mime: in
127         }
128
129         pmap := params
130         if idx := strings.Index(key, "*"); idx != -1
131             baseName := key[:idx]
132             if continuation == nil {
133                 continuation = make(map[stri
134             }
135             var ok bool
136             if pmap, ok = continuation[baseName]
137                 continuation[baseName] = mak
138                 pmap = continuation[baseName
139             }
140     }

```

```

141         if _, exists := pmap[key]; exists {
142             // Duplicate parameter name is bogus
143             return "", nil, errors.New("mime: du
144         }
145         pmap[key] = value
146         v = rest
147     }
148
149     // Stitch together any continuations or things with
150     // (i.e. RFC 2231 things with stars: "foo*0" or "foo
151     var buf bytes.Buffer
152     for key, pieceMap := range continuation {
153         singlePartKey := key + "*"
154         if v, ok := pieceMap[singlePartKey]; ok {
155             decv := decode2231Enc(v)
156             params[key] = decv
157             continue
158         }
159
160         buf.Reset()
161         valid := false
162         for n := 0; ; n++ {
163             simplePart := fmt.Sprintf("%s*%d", k
164             if v, ok := pieceMap[simplePart]; ok
165                 valid = true
166                 buf.WriteString(v)
167                 continue
168             }
169             encodedPart := simplePart + "*"
170             if v, ok := pieceMap[encodedPart]; o
171                 valid = true
172                 if n == 0 {
173                     buf.WriteString(deco
174                 } else {
175                     decv, _ := percentHe
176                     buf.WriteString(decv
177                 }
178             } else {
179                 break
180             }
181         }
182         if valid {
183             params[key] = buf.String()
184         }
185     }
186
187     return
188 }
189

```

```

190 func decode2231Enc(v string) string {
191     sv := strings.SplitN(v, "", 3)
192     if len(sv) != 3 {
193         return ""
194     }
195     // TODO: ignoring lang in sv[1] for now. If anybody
196     // need to decide how to expose it in the API. But I
197     // anybody uses it in practice.
198     charset := strings.ToLower(sv[0])
199     if charset != "us-ascii" && charset != "utf-8" {
200         // TODO: unsupported encoding
201         return ""
202     }
203     encv, _ := percentHexUnescape(sv[2])
204     return encv
205 }
206
207 func isNotTokenChar(r rune) bool {
208     return !isTokenChar(r)
209 }
210
211 // consumeToken consumes a token from the beginning of provi
212 // string, per RFC 2045 section 5.1 (referenced from 2183),
213 // the token consumed and the rest of the string. Returns (
214 // failure to consume at least one character.
215 func consumeToken(v string) (token, rest string) {
216     notPos := strings.IndexFunc(v, isNotTokenChar)
217     if notPos == -1 {
218         return v, ""
219     }
220     if notPos == 0 {
221         return "", v
222     }
223     return v[0:notPos], v[notPos:]
224 }
225
226 // consumeValue consumes a "value" per RFC 2045, where a val
227 // either a 'token' or a 'quoted-string'. On success, consu
228 // returns the value consumed (and de-quoted/escaped, if a
229 // quoted-string) and the rest of the string. On failure, r
230 // ("", v).
231 func consumeValue(v string) (value, rest string) {
232     if !strings.HasPrefix(v, `"` ) && !strings.HasPrefix(
233         return consumeToken(v)
234     }
235
236     leadQuote := rune(v[0])
237
238     // parse a quoted-string
239     rest = v[1:] // consume the leading quote

```

```

240     buffer := new(bytes.Buffer)
241     var idx int
242     var r rune
243     var nextIsLiteral bool
244     for idx, r = range rest {
245         switch {
246             case nextIsLiteral:
247                 buffer.WriteRune(r)
248                 nextIsLiteral = false
249             case r == leadQuote:
250                 return buffer.String(), rest[idx+1:]
251             case r == '\\':
252                 nextIsLiteral = true
253             case r != '\\r' && r != '\\n':
254                 buffer.WriteRune(r)
255             default:
256                 return "", v
257         }
258     }
259     return "", v
260 }
261
262 func consumeMediaParam(v string) (param, value, rest string)
263     rest = strings.TrimLeftFunc(v, unicode.IsSpace)
264     if !strings.HasPrefix(rest, ";") {
265         return "", "", v
266     }
267
268     rest = rest[1:] // consume semicolon
269     rest = strings.TrimLeftFunc(rest, unicode.IsSpace)
270     param, rest = consumeToken(rest)
271     param = strings.ToLower(param)
272     if param == "" {
273         return "", "", v
274     }
275
276     rest = strings.TrimLeftFunc(rest, unicode.IsSpace)
277     if !strings.HasPrefix(rest, "=") {
278         return "", "", v
279     }
280     rest = rest[1:] // consume equals sign
281     rest = strings.TrimLeftFunc(rest, unicode.IsSpace)
282     value, rest = consumeValue(rest)
283     if value == "" {
284         return "", "", v
285     }
286     return param, value, rest
287 }
288

```

```

289 func percentHexUnescape(s string) (string, error) {
290     // Count %, check that they're well-formed.
291     percents := 0
292     for i := 0; i < len(s); {
293         if s[i] != '%' {
294             i++
295             continue
296         }
297         percents++
298         if i+2 >= len(s) || !ishex(s[i+1]) || !ishex
299             s = s[i:]
300             if len(s) > 3 {
301                 s = s[0:3]
302             }
303             return "", fmt.Errorf("mime: bogus c
304         }
305         i += 3
306     }
307     if percents == 0 {
308         return s, nil
309     }
310
311     t := make([]byte, len(s)-2*percents)
312     j := 0
313     for i := 0; i < len(s); {
314         switch s[i] {
315             case '%':
316                 t[j] = unhex(s[i+1])<<4 | unhex(s[i+
317                     j++
318                     i += 3
319             default:
320                 t[j] = s[i]
321                 j++
322                 i++
323         }
324     }
325     return string(t), nil
326 }
327
328 func ishex(c byte) bool {
329     switch {
330     case '0' <= c && c <= '9':
331         return true
332     case 'a' <= c && c <= 'f':
333         return true
334     case 'A' <= c && c <= 'F':
335         return true
336     }
337     return false

```

```
338 }
339
340 func unhex(c byte) byte {
341     switch {
342     case '0' <= c && c <= '9':
343         return c - '0'
344     case 'a' <= c && c <= 'f':
345         return c - 'a' + 10
346     case 'A' <= c && c <= 'F':
347         return c - 'A' + 10
348     }
349     return 0
350 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/mime/type.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package mime implements parts of the MIME spec.
6 package mime
7
8 import (
9     "fmt"
10    "strings"
11    "sync"
12 )
13
14 var mimeTypeypes = map[string]string{
15     ".css": "text/css; charset=utf-8",
16     ".gif": "image/gif",
17     ".htm": "text/html; charset=utf-8",
18     ".html": "text/html; charset=utf-8",
19     ".jpg": "image/jpeg",
20     ".js": "application/x-javascript",
21     ".pdf": "application/pdf",
22     ".png": "image/png",
23     ".xml": "text/xml; charset=utf-8",
24 }
25
26 var mimeLock sync.RWMutex
27
28 var once sync.Once
29
30 // TypeByExtension returns the MIME type associated with the
31 // The extension ext should begin with a leading dot, as in
32 // When ext has no associated type, TypeByExtension returns
33 //
34 // The built-in table is small but on unix it is augmented b
35 // system's mime.types file(s) if available under one or mor
36 // names:
37 //
38 // /etc/mime.types
39 // /etc/apache2/mime.types
40 // /etc/apache/mime.types
41 //
42 // Windows system mime types are extracted from registry.
43 //
44 // Text types have the charset parameter set to "utf-8" by d
```

```

45 func TypeByExtension(ext string) string {
46     once.Do(initMime)
47     mimeLock.RLock()
48     typename := mimeTypes[ext]
49     mimeLock.RUnlock()
50     return typename
51 }
52
53 // AddExtensionType sets the MIME type associated with
54 // the extension ext to typ. The extension should begin wit
55 // a leading dot, as in ".html".
56 func AddExtensionType(ext, typ string) error {
57     if ext == "" || ext[0] != '.' {
58         return fmt.Errorf(`mime: extension "%s" miss
59     }
60     once.Do(initMime)
61     return setExtensionType(ext, typ)
62 }
63
64 func setExtensionType(extension, mimeType string) error {
65     _, param, err := ParseMediaType(mimeType)
66     if err != nil {
67         return err
68     }
69     if strings.HasPrefix(mimeType, "text/") && param["ch
70         param["charset"] = "utf-8"
71         mimeType = FormatMediaType(mimeType, param)
72     }
73     mimeLock.Lock()
74     mimeTypes[extension] = mimeType
75     mimeLock.Unlock()
76     return nil
77 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/mime/type_unix.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd plan9
6
7 package mime
8
9 import (
10     "bufio"
11     "os"
12     "strings"
13 )
14
15 var typeFiles = []string{
16     "/etc/mime.types",
17     "/etc/apache2/mime.types",
18     "/etc/apache/mime.types",
19 }
20
21 func loadMimeFile(filename string) {
22     f, err := os.Open(filename)
23     if err != nil {
24         return
25     }
26
27     reader := bufio.NewReader(f)
28     for {
29         line, err := reader.ReadString('\n')
30         if err != nil {
31             f.Close()
32             return
33         }
34         fields := strings.Fields(line)
35         if len(fields) <= 1 || fields[0][0] == '#' {
36             continue
37         }
38         mimeType := fields[0]
39         for _, ext := range fields[1:] {
40             if ext[0] == '#' {
41                 break

```

```
42             }
43             setExtensionType("."+ext, mimeType)
44         }
45     }
46 }
47
48 func initMime() {
49     for _, filename := range typeFiles {
50         loadMimeFile(filename)
51     }
52 }
53
54 func initMimeForTests() map[string]string {
55     typeFiles = []string{"test.types"}
56     return map[string]string{
57         ".t1": "application/test",
58         ".t2": "text/test; charset=utf-8",
59         ".png": "image/png",
60     }
61 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/mime/multipart/formdata.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package multipart
6
7 import (
8     "bytes"
9     "errors"
10    "io"
11    "io/ioutil"
12    "net/textproto"
13    "os"
14 )
15
16 // TODO(adg,bradfitz): find a way to unify the DoS-preventio
17 // with that of the http package's ParseForm.
18
19 // ReadForm parses an entire multipart message whose parts h
20 // a Content-Disposition of "form-data".
21 // It stores up to maxMemory bytes of the file parts in memo
22 // and the remainder on disk in temporary files.
23 func (r *Reader) ReadForm(maxMemory int64) (f *Form, err error) {
24     form := &Form{make(map[string][]string), make(map[string]
25     defer func() {
26         if err != nil {
27             form.RemoveAll()
28         }
29     }()
30
31     maxValueBytes := int64(10 << 20) // 10 MB is a lot o
32     for {
33         p, err := r.NextPart()
34         if err == io.EOF {
35             break
36         }
37         if err != nil {
38             return nil, err
39         }
40
41         name := p.FormName()
```

```

42         if name == "" {
43             continue
44         }
45         filename := p.FileName()
46
47         var b bytes.Buffer
48
49         if filename == "" {
50             // value, store as string in memory
51             n, err := io.CopyN(&b, p, maxValueBy
52             if err != nil && err != io.EOF {
53                 return nil, err
54             }
55             maxValueBytes -= n
56             if maxValueBytes == 0 {
57                 return nil, errors.New("mult
58             }
59             form.Value[name] = append(form.Value
60             continue
61         }
62
63         // file, store in memory or on disk
64         fh := &FileHeader{
65             Filename: filename,
66             Header:    p.Header,
67         }
68         n, err := io.CopyN(&b, p, maxMemory+1)
69         if err != nil && err != io.EOF {
70             return nil, err
71         }
72         if n > maxMemory {
73             // too big, write to disk and flush
74             file, err := ioutil.TempFile("", "mu
75             if err != nil {
76                 return nil, err
77             }
78             defer file.Close()
79             _, err = io.Copy(file, io.MultiReade
80             if err != nil {
81                 os.Remove(file.Name())
82                 return nil, err
83             }
84             fh.tmpfile = file.Name()
85         } else {
86             fh.content = b.Bytes()
87             maxMemory -= n
88         }
89         form.File[name] = append(form.File[name], fh
90     }
91

```

```

92         return form, nil
93     }
94
95     // Form is a parsed multipart form.
96     // Its File parts are stored either in memory or on disk,
97     // and are accessible via the *FileHeader's Open method.
98     // Its Value parts are stored as strings.
99     // Both are keyed by field name.
100    type Form struct {
101        Value map[string][]string
102        File  map[string][]*FileHeader
103    }
104
105    // RemoveAll removes any temporary files associated with a F
106    func (f *Form) RemoveAll() error {
107        var err error
108        for _, fhs := range f.File {
109            for _, fh := range fhs {
110                if fh.tmpfile != "" {
111                    e := os.Remove(fh.tmpfile)
112                    if e != nil && err == nil {
113                        err = e
114                    }
115                }
116            }
117        }
118        return err
119    }
120
121    // A FileHeader describes a file part of a multipart request
122    type FileHeader struct {
123        Filename string
124        Header   textproto.MIMEHeader
125
126        content []byte
127        tmpfile string
128    }
129
130    // Open opens and returns the FileHeader's associated File.
131    func (fh *FileHeader) Open() (File, error) {
132        if b := fh.content; b != nil {
133            r := io.NewSectionReader(bytes.NewReader(b),
134                sectionReadCloser{r}, nil
135        }
136        return os.Open(fh.tmpfile)
137    }
138
139    // File is an interface to access the file part of a multipa
140    // Its contents may be either stored in memory or on disk.

```

```
141 // If stored on disk, the File's underlying concrete type wi
142 type File interface {
143     io.Reader
144     io.ReaderAt
145     io.Seeker
146     io.Closer
147 }
148
149 // helper types to turn a []byte into a File
150
151 type sectionReadCloser struct {
152     *io.SectionReader
153 }
154
155 func (rc sectionReadCloser) Close() error {
156     return nil
157 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/mime/multipart/multipart.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4 //
5
6 /*
7 Package multipart implements MIME multipart parsing, as defini
8 2046.
9
10 The implementation is sufficient for HTTP (RFC 2388) and the
11 bodies generated by popular browsers.
12 */
13 package multipart
14
15 import (
16     "bufio"
17     "bytes"
18     "fmt"
19     "io"
20     "io/ioutil"
21     "mime"
22     "net/textproto"
23 )
24
25 // TODO(bradfitz): inline these once the compiler can inline
26 // read-only situation (such as bytes.HasSuffix)
27 var lf = []byte("\n")
28 var crlf = []byte("\r\n")
29
30 var emptyParams = make(map[string]string)
31
32 // A Part represents a single part in a multipart body.
33 type Part struct {
34     // The headers of the body, if any, with the keys ca
35     // in the same fashion that the Go http.Request head
36     // i.e. "foo-bar" changes case to "Foo-Bar"
37     Header textproto.MIMEHeader
38
39     buffer *bytes.Buffer
40     mr     *Reader
41 }
```

```

42         disposition      string
43         dispositionParams map[string]string
44     }
45
46     // FormName returns the name parameter if p has a Content-Di
47     // of type "form-data". Otherwise it returns the empty stri
48     func (p *Part) FormName() string {
49         // See http://tools.ietf.org/html/rfc2183 section 2
50         // of Content-Disposition value format.
51         if p.dispositionParams == nil {
52             p.parseContentDisposition()
53         }
54         if p.disposition != "form-data" {
55             return ""
56         }
57         return p.dispositionParams["name"]
58     }
59
60     // FileName returns the filename parameter of the Part's
61     // Content-Disposition header.
62     func (p *Part) FileName() string {
63         if p.dispositionParams == nil {
64             p.parseContentDisposition()
65         }
66         return p.dispositionParams["filename"]
67     }
68
69     func (p *Part) parseContentDisposition() {
70         v := p.Header.Get("Content-Disposition")
71         var err error
72         p.disposition, p.dispositionParams, err = mime.Parse
73         if err != nil {
74             p.dispositionParams = emptyParams
75         }
76     }
77
78     // NewReader creates a new multipart Reader reading from r u
79     // given MIME boundary.
80     func NewReader(reader io.Reader, boundary string) *Reader {
81         b := []byte("\r\n--" + boundary + "--")
82         return &Reader{
83             bufReader: bufio.NewReader(reader),
84
85             nl:             b[:2],
86             nlDashBoundary: b[:len(b)-2],
87             dashBoundaryDash: b[2:],
88             dashBoundary:   b[2 : len(b)-2],
89         }
90     }
91

```

```

92 func newPart(mr *Reader) (*Part, error) {
93     bp := &Part{
94         Header: make(map[string][]string),
95         mr:     mr,
96         buffer: new(bytes.Buffer),
97     }
98     if err := bp.populateHeaders(); err != nil {
99         return nil, err
100    }
101    return bp, nil
102 }
103
104 func (bp *Part) populateHeaders() error {
105     r := textproto.NewReader(bp.mr.bufReader)
106     header, err := r.ReadMIMEHeader()
107     if err == nil {
108         bp.Header = header
109     }
110     return err
111 }
112
113 // Read reads the body of a part, after its headers and before
114 // next part (if any) begins.
115 func (p *Part) Read(d []byte) (n int, err error) {
116     if p.buffer.Len() >= len(d) {
117         // Internal buffer of unconsumed data is larger
118         // than the read request. No need to parse more.
119         return p.buffer.Read(d)
120     }
121     peek, err := p.mr.bufReader.Peek(4096) // TODO(bradfitz)
122     unexpectedEof := err == io.EOF
123     if err != nil && !unexpectedEof {
124         return 0, fmt.Errorf("multipart: Part Read: %v", err)
125     }
126     if peek == nil {
127         panic("nil peek buf")
128     }
129
130     // Search the peek buffer for "\r\n--boundary". If found,
131     // consume everything up to the boundary. If not, consume
132     // as much of the peek buffer as cannot hold the boundary
133     // string.
134     nCopy := 0
135     foundBoundary := false
136     if idx := bytes.Index(peek, p.mr.nlDashBoundary); idx > 0 {
137         nCopy = idx
138         foundBoundary = true
139     } else if safeCount := len(peek) - len(p.mr.nlDashBoundary) > 0 {
140         nCopy = safeCount

```

```

141     } else if unexpectedEof {
142         // If we've run out of peek buffer and the b
143         // wasn't found (and can't possibly fit), we
144         // hit the end of the file unexpectedly.
145         return 0, io.ErrUnexpectedEOF
146     }
147     if nCopy > 0 {
148         if _, err := io.CopyN(p.buffer, p.mr.bufRead
149             return 0, err
150         }
151     }
152     n, err = p.buffer.Read(d)
153     if err == io.EOF && !foundBoundary {
154         // If the boundary hasn't been reached there
155         // read, so don't pass through an EOF from t
156         err = nil
157     }
158     return
159 }
160
161 func (p *Part) Close() error {
162     io.Copy(ioutil.Discard, p)
163     return nil
164 }
165
166 // Reader is an iterator over parts in a MIME multipart body
167 // Reader's underlying parser consumes its input as needed.
168 // isn't supported.
169 type Reader struct {
170     bufReader *bufio.Reader
171
172     currentPart *Part
173     partsRead   int
174
175     nl, nlDashBoundary, dashBoundaryDash, dashBoundary [
176 }
177
178 // NextPart returns the next part in the multipart or an err
179 // When there are no more parts, the error io.EOF is returne
180 func (r *Reader) NextPart() (*Part, error) {
181     if r.currentPart != nil {
182         r.currentPart.Close()
183     }
184
185     expectNewPart := false
186     for {
187         line, err := r.bufReader.ReadSlice('\n')
188         if err == io.EOF && bytes.Equal(line, r.dash
189             // If the buffer ends in "--boundary

```

```

190         // trailing "\r\n", ReadSlice will r
191         // (since it's missing the '\n'), bu
192         // multipart EOF so we need to retur
193         // a fmt-wrapped one.
194         return nil, io.EOF
195     }
196     if err != nil {
197         return nil, fmt.Errorf("multipart: N
198     }
199
200     if r.isBoundaryDelimiterLine(line) {
201         r.partsRead++
202         bp, err := newPart(r)
203         if err != nil {
204             return nil, err
205         }
206         r.currentPart = bp
207         return bp, nil
208     }
209
210     if hasPrefixThenNewline(line, r.dashBoundary
211         // Expected EOF
212         return nil, io.EOF
213     }
214
215     if expectNewPart {
216         return nil, fmt.Errorf("multipart: e
217     }
218
219     if r.partsRead == 0 {
220         // skip line
221         continue
222     }
223
224     // Consume the "\n" or "\r\n" separator betw
225     // body of the previous part and the boundar
226     // now expect will follow. (either a new par
227     // end boundary)
228     if bytes.Equal(line, r.nl) {
229         expectNewPart = true
230         continue
231     }
232
233     return nil, fmt.Errorf("multipart: unexpecte
234 }
235 panic("unreachable")
236 }
237
238 func (mr *Reader) isBoundaryDelimiterLine(line []byte) bool
239 // http://tools.ietf.org/html/rfc2046#section-5.1

```

```

240 // The boundary delimiter line is then defined as
241 // consisting entirely of two hyphen characters ("
242 // decimal value 45) followed by the boundary para
243 // value from the Content-Type header field, optio
244 // whitespace, and a terminating CRLF.
245 if !bytes.HasPrefix(line, mr.dashBoundary) {
246     return false
247 }
248 if bytes.HasSuffix(line, mr.nl) {
249     return onlyHorizontalWhitespace(line[len(mr.
250 )])
251 // Violate the spec and also support newlines withou
252 // carriage return...
253 if mr.partsRead == 0 && bytes.HasSuffix(line, lf) {
254     if onlyHorizontalWhitespace(line[len(mr.dash
255 )]) {
256         mr.nlDashBoundary = mr.nlDashBoundar
257         return true
258     }
259 }
260 return false
261 }
262
263 func onlyHorizontalWhitespace(s []byte) bool {
264     for _, b := range s {
265         if b != ' ' && b != '\t' {
266             return false
267         }
268     }
269     return true
270 }
271
272 func hasPrefixThenNewline(s, prefix []byte) bool {
273     return bytes.HasPrefix(s, prefix) &&
274         (len(s) == len(prefix)+1 && s[len(s)-1] == '
275         len(s) == len(prefix)+2 && bytes.Has
276 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/mime/multipart/writer.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package multipart
6
7 import (
8     "bytes"
9     "crypto/rand"
10    "errors"
11    "fmt"
12    "io"
13    "net/textproto"
14    "strings"
15 )
16
17 // A Writer generates multipart messages.
18 type Writer struct {
19     w          io.Writer
20     boundary  string
21     lastpart  *part
22 }
23
24 // NewWriter returns a new multipart writer with a random bo
25 // writing to w.
26 func NewWriter(w io.Writer) *Writer {
27     return &Writer{
28         w:          w,
29         boundary:  randomBoundary(),
30     }
31 }
32
33 // Boundary returns the Writer's randomly selected boundary
34 func (w *Writer) Boundary() string {
35     return w.boundary
36 }
37
38 // FormDataContentType returns the Content-Type for an HTTP
39 // multipart/form-data with this Writer's Boundary.
40 func (w *Writer) FormDataContentType() string {
41     return "multipart/form-data; boundary=" + w.boundary
```

```

42 }
43
44 func randomBoundary() string {
45     var buf [30]byte
46     _, err := io.ReadFull(rand.Reader, buf[:])
47     if err != nil {
48         panic(err)
49     }
50     return fmt.Sprintf("%x", buf[:])
51 }
52
53 // CreatePart creates a new multipart section with the provi
54 // header. The body of the part should be written to the ret
55 // Writer. After calling CreatePart, any previous part may n
56 // be written to.
57 func (w *Writer) CreatePart(header textproto.MIMEHeader) (io
58     if w.lastpart != nil {
59         if err := w.lastpart.close(); err != nil {
60             return nil, err
61         }
62     }
63     var b bytes.Buffer
64     if w.lastpart != nil {
65         fmt.Fprintf(&b, "\r\n--%s\r\n", w.boundary)
66     } else {
67         fmt.Fprintf(&b, "--%s\r\n", w.boundary)
68     }
69     // TODO(bradfitz): move this to textproto.MimeHeader
70     // and clean, like http.Header.Write(w) does.
71     for k, vv := range header {
72         for _, v := range vv {
73             fmt.Fprintf(&b, "%s: %s\r\n", k, v)
74         }
75     }
76     fmt.Fprintf(&b, "\r\n")
77     _, err := io.Copy(w.w, &b)
78     if err != nil {
79         return nil, err
80     }
81     p := &part{
82         mw: w,
83     }
84     w.lastpart = p
85     return p, nil
86 }
87
88 var quoteEscaper = strings.NewReplacer("\\", "\\\\", `\"`, "\\`")
89
90 func escapeQuotes(s string) string {
91     return quoteEscaper.Replace(s)

```

```

92 }
93
94 // CreateFormFile is a convenience wrapper around CreatePart
95 // a new form-data header with the provided field name and f
96 func (w *Writer) CreateFormFile(fieldname, filename string)
97     h := make(textproto.MIMEHeader)
98     h.Set("Content-Disposition",
99         fmt.Sprintf(`form-data; name="%s"; filename=
100             escapeQuotes(fieldname), escapeQuote
101     h.Set("Content-Type", "application/octet-stream")
102     return w.CreatePart(h)
103 }
104
105 // CreateFormField calls CreatePart with a header using the
106 // given field name.
107 func (w *Writer) CreateFormField(fieldname string) (io.Writer
108     h := make(textproto.MIMEHeader)
109     h.Set("Content-Disposition",
110         fmt.Sprintf(`form-data; name="%s"`, escapeQu
111     return w.CreatePart(h)
112 }
113
114 // WriteField calls CreateFormField and then writes the give
115 func (w *Writer) WriteField(fieldname, value string) error {
116     p, err := w.CreateFormField(fieldname)
117     if err != nil {
118         return err
119     }
120     _, err = p.Write([]byte(value))
121     return err
122 }
123
124 // Close finishes the multipart message and writes the trail
125 // boundary end line to the output.
126 func (w *Writer) Close() error {
127     if w.lastpart != nil {
128         if err := w.lastpart.close(); err != nil {
129             return err
130         }
131         w.lastpart = nil
132     }
133     _, err := fmt.Fprintf(w.w, "\r\n--%s--\r\n", w.bound
134     return err
135 }
136
137 type part struct {
138     mw      *Writer
139     closed bool
140     we      error // last error that occurred writing

```

```
141 }
142
143 func (p *part) close() error {
144     p.closed = true
145     return p.we
146 }
147
148 func (p *part) Write(d []byte) (n int, err error) {
149     if p.closed {
150         return 0, errors.New("multipart: can't write
151     }
152     n, err = p.mw.w.Write(d)
153     if err != nil {
154         p.we = err
155     }
156     return
157 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/cgo_linux.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package net
6
7 /*
8 #include <netdb.h>
9 */
10 import "C"
11
12 func cgoAddrInfoMask() C.int {
13     return C.AI_CANONNAME | C.AI_V4MAPPED | C.AI_ALL
14 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/cgo_unix.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux
6
7 package net
8
9 /*
10 #include <sys/types.h>
11 #include <sys/socket.h>
12 #include <netinet/in.h>
13 #include <netdb.h>
14 #include <stdlib.h>
15 #include <unistd.h>
16 #include <string.h>
17 */
18 import "C"
19
20 import (
21     "syscall"
22     "unsafe"
23 )
24
25 func cgoLookupHost(name string) (addrs []string, err error,
26     ip, err, completed := cgoLookupIP(name)
27     for _, p := range ip {
28         addrs = append(addrs, p.String())
29     }
30     return
31 }
32
33 func cgoLookupPort(net, service string) (port int, err error
34     var res *C.struct_addrinfo
35     var hints C.struct_addrinfo
36
37     switch net {
38     case "":
39         // no hints
40     case "tcp", "tcp4", "tcp6":
41         hints.ai_socktype = C.SOCK_STREAM
42         hints.ai_protocol = C.IPPROTO_TCP
43     case "udp", "udp4", "udp6":
44         hints.ai_socktype = C.SOCK_DGRAM
```

```

45         hints.ai_protocol = C.IPPROTO_UDP
46     default:
47         return 0, UnknownNetworkError(net), true
48     }
49     if len(net) >= 4 {
50         switch net[3] {
51             case '4':
52                 hints.ai_family = C.AF_INET
53             case '6':
54                 hints.ai_family = C.AF_INET6
55         }
56     }
57
58     s := C.CString(service)
59     defer C.free(unsafe.Pointer(s))
60     if C.getaddrinfo(nil, s, &hints, &res) == 0 {
61         defer C.freeaddrinfo(res)
62         for r := res; r != nil; r = r.ai_next {
63             switch r.ai_family {
64                 default:
65                     continue
66                 case C.AF_INET:
67                     sa := (*syscall.RawSockaddrI
68                     p := (*[2]byte)(unsafe.Pointer(
69                     return int(p[0])<<8 | int(p[
70                 case C.AF_INET6:
71                     sa := (*syscall.RawSockaddrI
72                     p := (*[2]byte)(unsafe.Pointer(
73                     return int(p[0])<<8 | int(p[
74             }
75         }
76     }
77     return 0, &AddrError{"unknown port", net + "/" + ser
78 }
79
80 func cgoLookupIPCNAM(name string) (addrs []IP, cname string
81     var res *C.struct_addrinfo
82     var hints C.struct_addrinfo
83
84     // NOTE(rsc): In theory there are approximately bala
85     // arguments for and against including AI_ADDRCONFIG
86     // in the flags (it includes IPv4 results only on IP
87     // and similarly for IPv6), but in practice setting
88     // getaddrinfo to return the wrong canonical name on
89     // So definitely leave it out.
90     hints.ai_flags = (C.AI_ALL | C.AI_V4MAPPED | C.AI_CA
91
92     h := C.CString(name)
93     defer C.free(unsafe.Pointer(h))
94     gerrno, err := C.getaddrinfo(h, nil, &hints, &res)

```

```

95     if gerrno != 0 {
96         var str string
97         if gerrno == C.EAI_NONAME {
98             str = noSuchHost
99         } else if gerrno == C.EAI_SYSTEM {
100            str = err.Error()
101        } else {
102            str = C.GoString(C.gai_strerror(gerr
103        })
104        return nil, "", &DNSError{Err: str, Name: na
105    }
106    defer C.freeaddrinfo(res)
107    if res != nil {
108        cname = C.GoString(res.ai_canonname)
109        if cname == "" {
110            cname = name
111        }
112        if len(cname) > 0 && cname[len(cname)-1] !=
113            cname += "."
114        }
115    }
116    for r := res; r != nil; r = r.ai_next {
117        // Everything comes back twice, once for UDP
118        if r.ai_socktype != C.SOCK_STREAM {
119            continue
120        }
121        switch r.ai_family {
122        default:
123            continue
124        case C.AF_INET:
125            sa := (*syscall.RawSockaddrInet4)(un
126            addr = append(addr, copyIP(sa.Addr
127        case C.AF_INET6:
128            sa := (*syscall.RawSockaddrInet6)(un
129            addr = append(addr, copyIP(sa.Addr
130        }
131    }
132    return addr, cname, nil, true
133 }
134
135 func cgoLookupIP(name string) (addr []IP, err error, comple
136     addr, _, err, completed = cgoLookupIPcname(name)
137     return
138 }
139
140 func cgoLookupCNAME(name string) (cname string, err error, c
141     _, cname, err, completed = cgoLookupIPcname(name)
142     return
143 }

```

```
144
145 func copyIP(x IP) IP {
146     y := make(IP, len(x))
147     copy(y, x)
148     return y
149 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/dial.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package net
6
7 import (
8     "time"
9 )
10
11 func parseDialNetwork(net string) (afnet string, proto int,
12     i := last(net, ':')
13     if i < 0 { // no colon
14         switch net {
15             case "tcp", "tcp4", "tcp6":
16             case "udp", "udp4", "udp6":
17             case "unix", "unixgram", "unixpacket":
18             default:
19                 return "", 0, UnknownNetworkError(ne
20             }
21             return net, 0, nil
22         }
23         afnet = net[:i]
24         switch afnet {
25             case "ip", "ip4", "ip6":
26                 protostr := net[i+1:]
27                 proto, i, ok := dtoi(protostr, 0)
28                 if !ok || i != len(protostr) {
29                     proto, err = lookupProtocol(protostr)
30                     if err != nil {
31                         return "", 0, err
32                     }
33                 }
34                 return afnet, proto, nil
35             }
36             return "", 0, UnknownNetworkError(net)
37         }
38
39 func resolveNetAddr(op, net, addr string) (afnet string, a A
40     afnet, _, err = parseDialNetwork(net)
41     if err != nil {
42         return "", nil, &OpError{op, net, nil, err}
43     }
44     if op == "dial" && addr == "" {
```

```

45         return "", nil, &OpError{op, net, nil, errMi
46     }
47     switch afnet {
48     case "tcp", "tcp4", "tcp6":
49         if addr != "" {
50             a, err = ResolveTCPAddr(afnet, addr)
51         }
52     case "udp", "udp4", "udp6":
53         if addr != "" {
54             a, err = ResolveUDPAddr(afnet, addr)
55         }
56     case "ip", "ip4", "ip6":
57         if addr != "" {
58             a, err = ResolveIPAddr(afnet, addr)
59         }
60     case "unix", "unixgram", "unixpacket":
61         if addr != "" {
62             a, err = ResolveUnixAddr(afnet, addr
63         }
64     }
65     return
66 }
67
68 // Dial connects to the address addr on the network net.
69 //
70 // Known networks are "tcp", "tcp4" (IPv4-only), "tcp6" (IPv
71 // "udp", "udp4" (IPv4-only), "udp6" (IPv6-only), "ip", "ip4
72 // (IPv4-only), "ip6" (IPv6-only), "unix" and "unixpacket".
73 //
74 // For TCP and UDP networks, addresses have the form host:po
75 // If host is a literal IPv6 address, it must be enclosed
76 // in square brackets. The functions JoinHostPort and Split
77 // manipulate addresses in this form.
78 //
79 // Examples:
80 //     Dial("tcp", "12.34.56.78:80")
81 //     Dial("tcp", "google.com:80")
82 //     Dial("tcp", "[de:ad:be:ef::ca:fe]:80")
83 //
84 // For IP networks, addr must be "ip", "ip4" or "ip6" follow
85 // by a colon and a protocol number or name.
86 //
87 // Examples:
88 //     Dial("ip4:1", "127.0.0.1")
89 //     Dial("ip6:ospf", ":::1")
90 //
91 func Dial(net, addr string) (Conn, error) {
92     _, addr1, err := resolveNetAddr("dial", net, addr)
93     if err != nil {
94         return nil, err

```

```

95     }
96     return dialAddr(net, addr, addri)
97 }
98
99 func dialAddr(net, addr string, addri Addr) (c Conn, err error) {
100     switch ra := addri.(type) {
101     case *TCPAddr:
102         c, err = DialTCP(net, nil, ra)
103     case *UDPAddr:
104         c, err = DialUDP(net, nil, ra)
105     case *IPAddr:
106         c, err = DialIP(net, nil, ra)
107     case *UnixAddr:
108         c, err = DialUnix(net, nil, ra)
109     default:
110         err = &OpError{"dial", net + " " + addr, nil}
111     }
112     if err != nil {
113         return nil, err
114     }
115     return
116 }
117
118 // DialTimeout acts like Dial but takes a timeout.
119 // The timeout includes name resolution, if required.
120 func DialTimeout(net, addr string, timeout time.Duration) (C
121     // TODO(bradfitz): the timeout should be pushed down
122     // net package's event loop, so on timeout to dead h
123     // don't have a goroutine sticking around for the de
124     // ~3 minutes.
125     t := time.NewTimer(timeout)
126     defer t.Stop()
127     type pair struct {
128         Conn
129         error
130     }
131     ch := make(chan pair, 1)
132     resolvedAddr := make(chan Addr, 1)
133     go func() {
134         _, addri, err := resolveNetAddr("dial", net,
135         if err != nil {
136             ch <- pair{nil, err}
137             return
138         }
139         resolvedAddr <- addri // in case we need it
140         c, err := dialAddr(net, addr, addri)
141         ch <- pair{c, err}
142     }()
143     select {

```

```

144     case <-t.C:
145         // Try to use the real Addr in our OpError,
146         // before the timeout. Otherwise we just use
147         var addri Addr
148         select {
149         case a := <-resolvedAddr:
150             addri = a
151         default:
152             addri = &stringAddr{net, addr}
153         }
154         err := &OpError{
155             Op: "dial",
156             Net: net,
157             Addr: addri,
158             Err: &timeoutError{},
159         }
160         return nil, err
161     case p := <-ch:
162         return p.Conn, p.error
163     }
164     panic("unreachable")
165 }
166
167 type stringAddr struct {
168     net, addr string
169 }
170
171 func (a stringAddr) Network() string { return a.net }
172 func (a stringAddr) String() string { return a.addr }
173
174 // Listen announces on the local network address laddr.
175 // The network string net must be a stream-oriented network:
176 // "tcp", "tcp4", "tcp6", or "unix", or "unixpacket".
177 func Listen(net, laddr string) (Listener, error) {
178     afnet, a, err := resolveNetAddr("listen", net, laddr)
179     if err != nil {
180         return nil, err
181     }
182     switch afnet {
183     case "tcp", "tcp4", "tcp6":
184         var la *TCPAddr
185         if a != nil {
186             la = a.(*TCPAddr)
187         }
188         return ListenTCP(net, la)
189     case "unix", "unixpacket":
190         var la *UnixAddr
191         if a != nil {
192             la = a.(*UnixAddr)

```

```

193         }
194         return ListenUnix(net, la)
195     }
196     return nil, UnknownNetworkError(net)
197 }
198
199 // ListenPacket announces on the local network address laddr
200 // The network string net must be a packet-oriented network:
201 // "udp", "udp4", "udp6", "ip", "ip4", "ip6" or "unixgram".
202 func ListenPacket(net, addr string) (PacketConn, error) {
203     afnet, a, err := resolveNetAddr("listen", net, addr)
204     if err != nil {
205         return nil, err
206     }
207     switch afnet {
208     case "udp", "udp4", "udp6":
209         var la *UDPAddr
210         if a != nil {
211             la = a.(*UDPAddr)
212         }
213         return ListenUDP(net, la)
214     case "ip", "ip4", "ip6":
215         var la *IPAddr
216         if a != nil {
217             la = a.(*IPAddr)
218         }
219         return ListenIP(net, la)
220     case "unixgram":
221         var la *UnixAddr
222         if a != nil {
223             la = a.(*UnixAddr)
224         }
225         return DialUnix(net, la, nil)
226     }
227     return nil, UnknownNetworkError(net)
228 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/dnsclient.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package net
6
7 import (
8     "math/rand"
9     "sort"
10 )
11
12 // DNSError represents a DNS lookup error.
13 type DNSError struct {
14     Err        string // description of the error
15     Name       string // name looked for
16     Server     string // server used
17     IsTimeout  bool
18 }
19
20 func (e *DNSError) Error() string {
21     if e == nil {
22         return "<nil>"
23     }
24     s := "lookup " + e.Name
25     if e.Server != "" {
26         s += " on " + e.Server
27     }
28     s += ": " + e.Err
29     return s
30 }
31
32 func (e *DNSError) Timeout() bool { return e.IsTimeout }
33 func (e *DNSError) Temporary() bool { return e.IsTimeout }
34
35 const noSuchHost = "no such host"
36
37 // reverseaddr returns the in-addr.arpa. or ip6.arpa. hostna
38 // address addr suitable for rDNS (PTR) record lookup or an
39 // to parse the IP address.
40 func reverseaddr(addr string) (arpa string, err error) {
41     ip := ParseIP(addr)
42     if ip == nil {
43         return "", &DNSError{Err: "unrecognized addr"}
44     }
45 }
```

```

45     if ip.To4() != nil {
46         return itoa(int(ip[15])) + "." + itoa(int(ip
47             itoa(int(ip[12])) + ".in-addr.arpa."
48     }
49     // Must be IPv6
50     buf := make([]byte, 0, len(ip)*4+len("ip6.arpa."))
51     // Add it, in reverse, to the buffer
52     for i := len(ip) - 1; i >= 0; i-- {
53         v := ip[i]
54         buf = append(buf, hexDigit[v&0xF])
55         buf = append(buf, '.')
56         buf = append(buf, hexDigit[v>>4])
57         buf = append(buf, '.')
58     }
59     // Append "ip6.arpa." and return (buf already has th
60     buf = append(buf, "ip6.arpa."...)
61     return string(buf), nil
62 }
63
64 // Find answer for name in dns message.
65 // On return, if err == nil, addrs != nil.
66 func answer(name, server string, dns *dnsMsg, qtype uint16)
67     addrs = make([]dnsRR, 0, len(dns.answer))
68
69     if dns.rcode == dnsRcodeNameError && dns.recursion_a
70         return "", nil, &DNSError{Err: noSuchHost, N
71     }
72     if dns.rcode != dnsRcodeSuccess {
73         // None of the error codes make sense
74         // for the query we sent. If we didn't get
75         // a name error and we didn't get success,
76         // the server is behaving incorrectly.
77         return "", nil, &DNSError{Err: "server misbe
78     }
79
80     // Look for the name.
81     // Presotto says it's okay to assume that servers li
82     // /etc/resolv.conf are recursive resolvers.
83     // We asked for recursion, so it should have include
84     // all the answers we need in this one packet.
85     Cname:
86     for cnameloop := 0; cnameloop < 10; cnameloop++ {
87         addrs = addrs[0:0]
88         for _, rr := range dns.answer {
89             if _, justHeader := rr.(*dnsRR_Heade
90                 // Corrupt record: we only h
91                 // header. That header might
92                 // of type qtype, but we don
93                 // actually have it. Skip.
94                 continue

```

```

95         }
96         h := rr.Header()
97         if h.Class == dnsClassINET && h.Name
98             switch h.Rrtype {
99                 case qtype:
100                     addr = append(addr)
101                 case dnsTypeCNAME:
102                     // redirect to cname
103                     name = rr.(*dnsRR_CN
104                         continue Cname
105             }
106         }
107     }
108     if len(addr) == 0 {
109         return "", nil, &DNSError{Err: noSuc
110     }
111     return name, addr, nil
112 }
113
114 return "", nil, &DNSError{Err: "too many redirects",
115 }
116
117 func isDomainName(s string) bool {
118     // See RFC 1035, RFC 3696.
119     if len(s) == 0 {
120         return false
121     }
122     if len(s) > 255 {
123         return false
124     }
125     if s[len(s)-1] != '.' { // simplify checking loop: r
126         s += "."
127     }
128
129     last := byte('.')
130     ok := false // ok once we've seen a letter
131     partlen := 0
132     for i := 0; i < len(s); i++ {
133         c := s[i]
134         switch {
135             default:
136                 return false
137             case 'a' <= c && c <= 'z' || 'A' <= c && c <
138                 ok = true
139                 partlen++
140             case '0' <= c && c <= '9':
141                 // fine
142                 partlen++
143             case c == '-':

```

```

144         // byte before dash cannot be dot
145         if last == '.' {
146             return false
147         }
148         partlen++
149     case c == '.':
150         // byte before dot cannot be dot, da
151         if last == '.' || last == '-' {
152             return false
153         }
154         if partlen > 63 || partlen == 0 {
155             return false
156         }
157         partlen = 0
158     }
159     last = c
160 }
161
162     return ok
163 }
164
165 // An SRV represents a single DNS SRV record.
166 type SRV struct {
167     Target    string
168     Port      uint16
169     Priority  uint16
170     Weight    uint16
171 }
172
173 // byPriorityWeight sorts SRV records by ascending priority
174 type byPriorityWeight []*SRV
175
176 func (s byPriorityWeight) Len() int { return len(s) }
177
178 func (s byPriorityWeight) Swap(i, j int) { s[i], s[j] = s[j] }
179
180 func (s byPriorityWeight) Less(i, j int) bool {
181     return s[i].Priority < s[j].Priority ||
182         (s[i].Priority == s[j].Priority && s[i].Weig
183 }
184
185 // shuffleByWeight shuffles SRV records by weight using the
186 // described in RFC 2782.
187 func (addrs byPriorityWeight) shuffleByWeight() {
188     sum := 0
189     for _, addr := range addrs {
190         sum += int(addr.Weight)
191     }
192     for sum > 0 && len(addrs) > 1 {

```

```

193         s := 0
194         n := rand.Intn(sum + 1)
195         for i := range addrs {
196             s += int(addrs[i].Weight)
197             if s >= n {
198                 if i > 0 {
199                     t := addrs[i]
200                     copy(addrs[1:i+1], a)
201                     addrs[0] = t
202                 }
203                 break
204             }
205         }
206         sum -= int(addrs[0].Weight)
207         addrs = addrs[1:]
208     }
209 }
210
211 // sort reorders SRV records as specified in RFC 2782.
212 func (addrs byPriorityWeight) sort() {
213     sort.Sort(addrs)
214     i := 0
215     for j := 1; j < len(addrs); j++ {
216         if addrs[i].Priority != addrs[j].Priority {
217             addrs[i:j].shuffleByWeight()
218             i = j
219         }
220     }
221     addrs[i:].shuffleByWeight()
222 }
223
224 // An MX represents a single DNS MX record.
225 type MX struct {
226     Host string
227     Pref uint16
228 }
229
230 // byPref implements sort.Interface to sort MX records by pr
231 type byPref []*MX
232
233 func (s byPref) Len() int { return len(s) }
234
235 func (s byPref) Less(i, j int) bool { return s[i].Pref < s[j]
236
237 func (s byPref) Swap(i, j int) { s[i], s[j] = s[j], s[i] }
238
239 // sort reorders MX records as specified in RFC 5321.
240 func (s byPref) sort() {
241     for i := range s {
242         j := rand.Intn(i + 1)

```

```
243             s[i], s[j] = s[j], s[i]
244         }
245     sort.Sort(s)
246 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/dnsclient_unix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 // DNS client: see RFC 1035.
8 // Has to be linked into package net for Dial.
9
10 // TODO(rsc):
11 //     Check periodically whether /etc/resolv.conf has chan
12 //     Could potentially handle many outstanding lookups fa
13 //     Could have a small cache.
14 //     Random UDP source port (net.Dial should do that for
15 //     Random request IDs.
16
17 package net
18
19 import (
20     "math/rand"
21     "sync"
22     "time"
23 )
24
25 // Send a request on the connection and hope for a reply.
26 // Up to cfg.attempts attempts.
27 func exchange(cfg *dnsConfig, c Conn, name string, qtype uin
28     if len(name) >= 256 {
29         return nil, &DNSError{Err: "name too long",
30     }
31     out := new(dnsMsg)
32     out.id = uint16(rand.Int()) ^ uint16(time.Now().Unix
33     out.question = []dnsQuestion{
34         {name, qtype, dnsClassINET},
35     }
36     out.recursion_desired = true
37     msg, ok := out.Pack()
38     if !ok {
39         return nil, &DNSError{Err: "internal error -
40     }
41
```

```

42     for attempt := 0; attempt < cfg.attempts; attempt++
43         n, err := c.Write(msg)
44         if err != nil {
45             return nil, err
46         }
47
48         if cfg.timeout == 0 {
49             c.SetReadDeadline(time.Time{})
50         } else {
51             c.SetReadDeadline(time.Now().Add(time.Duration(cfg.timeout)))
52         }
53
54         buf := make([]byte, 2000) // More than enough
55         n, err = c.Read(buf)
56         if err != nil {
57             if e, ok := err.(Error); ok && e.Timeout() {
58                 continue
59             }
60             return nil, err
61         }
62         buf = buf[0:n]
63         in := new(dnsMsg)
64         if !in.Unpack(buf) || in.id != out.id {
65             continue
66         }
67         return in, nil
68     }
69     var server string
70     if a := c.RemoteAddr(); a != nil {
71         server = a.String()
72     }
73     return nil, &DNSError{Err: "no answer from server",
74 }
75
76 // Do a lookup for a single name, which must be rooted
77 // (otherwise answer will not find the answers).
78 func tryOneName(cfg *dnsConfig, name string, qtype uint16) (
79     if len(cfg.servers) == 0 {
80         return "", nil, &DNSError{Err: "no DNS server"}
81     }
82     for i := 0; i < len(cfg.servers); i++ {
83         // Calling Dial here is scary -- we have to
84         // not to dial a name that will require a DNS
85         // or Dial will call back here to translate
86         // The DNS config parser has already checked
87         // all the cfg.servers[i] are IP addresses,
88         // Dial will use without a DNS lookup.
89         server := cfg.servers[i] + ":53"
90         c, cerr := Dial("udp", server)
91         if cerr != nil {

```

```

92             err = cerr
93             continue
94         }
95         msg, merr := exchange(cfg, c, name, qtype)
96         c.Close()
97         if merr != nil {
98             err = merr
99             continue
100        }
101        cname, addrs, err = answer(name, server, msg)
102        if err == nil || err.(*DNSError).Err == noSu
103            break
104        }
105    }
106    return
107 }
108
109 func convertRR_A(records []dnsRR) []IP {
110     addrs := make([]IP, len(records))
111     for i, rr := range records {
112         a := rr.(*dnsRR_A).A
113         addrs[i] = IPv4(byte(a>>24), byte(a>>16), by
114     }
115     return addrs
116 }
117
118 func convertRR_AAAA(records []dnsRR) []IP {
119     addrs := make([]IP, len(records))
120     for i, rr := range records {
121         a := make(IP, IPv6len)
122         copy(a, rr.(*dnsRR_AAAA).AAAA[:])
123         addrs[i] = a
124     }
125     return addrs
126 }
127
128 var cfg *dnsConfig
129 var dnserr error
130
131 func loadConfig() { cfg, dnserr = dnsReadConfig() }
132
133 var onceLoadConfig sync.Once
134
135 func lookup(name string, qtype uint16) (cname string, addrs
136     if !isDomainName(name) {
137         return name, nil, &DNSError{Err: "invalid do
138     }
139     onceLoadConfig.Do(loadConfig)
140     if dnserr != nil || cfg == nil {

```

```

141         err = dnserr
142         return
143     }
144     // If name is rooted (trailing dot) or has enough do
145     // try it by itself first.
146     rooted := len(name) > 0 && name[len(name)-1] == '.'
147     if rooted || count(name, '.') >= cfg.ndots {
148         rname := name
149         if !rooted {
150             rname += "."
151         }
152         // Can try as ordinary name.
153         cname, addrs, err = tryOneName(cfg, rname, q)
154         if err == nil {
155             return
156         }
157     }
158     if rooted {
159         return
160     }
161
162     // Otherwise, try suffixes.
163     for i := 0; i < len(cfg.search); i++ {
164         rname := name + "." + cfg.search[i]
165         if rname[len(rname)-1] != '.' {
166             rname += "."
167         }
168         cname, addrs, err = tryOneName(cfg, rname, q)
169         if err == nil {
170             return
171         }
172     }
173
174     // Last ditch effort: try unsuffixed.
175     rname := name
176     if !rooted {
177         rname += "."
178     }
179     cname, addrs, err = tryOneName(cfg, rname, qtype)
180     if err == nil {
181         return
182     }
183     return
184 }
185
186 // goLookupHost is the native Go implementation of LookupHost
187 // Used only if cgoLookupHost refuses to handle the request
188 // (that is, only if cgoLookupHost is the stub in cgo_stub.g
189 // Normally we let cgo use the C library resolver instead of

```

```

190 // depending on our lookup code, so that Go and C get the sa
191 // answers.
192 func goLookupHost(name string) (addrs []string, err error) {
193     // Use entries from /etc/hosts if they match.
194     addrs = lookupStaticHost(name)
195     if len(addrs) > 0 {
196         return
197     }
198     onceLoadConfig.Do(loadConfig)
199     if dnserr != nil || cfg == nil {
200         err = dnserr
201         return
202     }
203     ips, err := goLookupIP(name)
204     if err != nil {
205         return
206     }
207     addrs = make([]string, 0, len(ips))
208     for _, ip := range ips {
209         addrs = append(addrs, ip.String())
210     }
211     return
212 }
213
214 // goLookupIP is the native Go implementation of LookupIP.
215 // Used only if cgoLookupIP refuses to handle the request
216 // (that is, only if cgoLookupIP is the stub in cgo_stub.go)
217 // Normally we let cgo use the C library resolver instead of
218 // depending on our lookup code, so that Go and C get the sa
219 // answers.
220 func goLookupIP(name string) (addrs []IP, err error) {
221     // Use entries from /etc/hosts if possible.
222     haddrs := lookupStaticHost(name)
223     if len(haddrs) > 0 {
224         for _, haddr := range haddrs {
225             if ip := ParseIP(haddr); ip != nil {
226                 addrs = append(addrs, ip)
227             }
228         }
229         if len(addrs) > 0 {
230             return
231         }
232     }
233     onceLoadConfig.Do(loadConfig)
234     if dnserr != nil || cfg == nil {
235         err = dnserr
236         return
237     }
238     var records []dnsRR
239     var cname string

```

```

240     cname, records, err = lookup(name, dnsTypeA)
241     if err != nil {
242         return
243     }
244     addrs = convertRR_A(records)
245     if cname != "" {
246         name = cname
247     }
248     _, records, err = lookup(name, dnsTypeAAAA)
249     if err != nil && len(addrs) > 0 {
250         // Ignore error because A lookup succeeded.
251         err = nil
252     }
253     if err != nil {
254         return
255     }
256     addrs = append(addrs, convertRR_AAAA(records)...)
257     return
258 }
259
260 // goLookupCNAME is the native Go implementation of LookupCN
261 // Used only if cgoLookupCNAME refuses to handle the request
262 // (that is, only if cgoLookupCNAME is the stub in cgo_stub.
263 // Normally we let cgo use the C library resolver instead of
264 // depending on our lookup code, so that Go and C get the sa
265 // answers.
266 func goLookupCNAME(name string) (cname string, err error) {
267     onceLoadConfig.Do(loadConfig)
268     if dnserr != nil || cfg == nil {
269         err = dnserr
270         return
271     }
272     _, rr, err := lookup(name, dnsTypeCNAME)
273     if err != nil {
274         return
275     }
276     cname = rr[0].(*dnsRR_CNAME).Cname
277     return
278 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/dnsconfig.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 // Read system DNS config from /etc/resolv.conf
8
9 package net
10
11 type dnsConfig struct {
12     servers []string // servers to use
13     search  []string // suffixes to append to local name
14     ndots   int      // number of dots in name to trigger search
15     timeout int      // seconds before giving up on packet
16     attempts int     // lost packets before giving up on connection
17     rotate  bool     // round robin among servers
18 }
19
20 // See resolv.conf(5) on a Linux machine.
21 // TODO(rsc): Supposed to call uname() and chop the beginning
22 // of the host name to get the default search domain.
23 // We assume it's in resolv.conf anyway.
24 func dnsReadConfig() (*dnsConfig, error) {
25     file, err := open("/etc/resolv.conf")
26     if err != nil {
27         return nil, &DNSConfigError{err}
28     }
29     conf := new(dnsConfig)
30     conf.servers = make([]string, 3)[0:0] // small, but
31     conf.search = make([]string, 0)
32     conf.ndots = 1
33     conf.timeout = 5
34     conf.attempts = 2
35     conf.rotate = false
36     for line, ok := file.readLine(); ok; line, ok = file
37         f := getFields(line)
38         if len(f) < 1 {
39             continue
40         }
41         switch f[0] {
42         case "nameserver": // add one name server
43             a := conf.servers
44             n := len(a)
```

```

45         if len(f) > 1 && n < cap(a) {
46             // One more check: make sure
47             // just an IP address. Othe
48             // to look it up.
49             name := f[1]
50             switch len(ParseIP(name)) {
51             case 16:
52                 name = "[" + name +
53                 fallthrough
54             case 4:
55                 a = a[0 : n+1]
56                 a[n] = name
57                 conf.servers = a
58             }
59         }
60
61     case "domain": // set search path to just th
62         if len(f) > 1 {
63             conf.search = make([]string,
64             conf.search[0] = f[1]
65         } else {
66             conf.search = make([]string,
67         }
68
69     case "search": // set search path to given s
70         conf.search = make([]string, len(f)-
71         for i := 0; i < len(conf.search); i+
72             conf.search[i] = f[i+1]
73         }
74
75     case "options": // magic options
76         for i := 1; i < len(f); i++ {
77             s := f[i]
78             switch {
79             case len(s) >= 6 && s[0:6] =
80                 n, _, _ := dtoi(s, 6
81                 if n < 1 {
82                     n = 1
83                 }
84                 conf.ndots = n
85             case len(s) >= 8 && s[0:8] =
86                 n, _, _ := dtoi(s, 8
87                 if n < 1 {
88                     n = 1
89                 }
90                 conf.timeout = n
91             case len(s) >= 9 && s[0:9] =
92                 n, _, _ := dtoi(s, 9
93                 if n < 1 {
94                     n = 1

```

```

95                                     }
96                                     conf.attempts = n
97     case s == "rotate":
98         conf.rotate = true
99     }
100                                     }
101     }
102 }
103     file.close()
104
105     return conf, nil
106 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/dnsmsg.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // DNS packet assembly. See RFC 1035.
6 //
7 // This is intended to support name resolution during Dial.
8 // It doesn't have to be blazing fast.
9 //
10 // Each message structure has a Walk method that is used by
11 // a generic pack/unpack routine. Thus, if in the future we
12 // to define new message structs, no new pack/unpack/printin
13 // needs to be written.
14 //
15 // The first half of this file defines the DNS message forma
16 // The second half implements the conversion to and from wir
17 // A few of the structure elements have string tags to aid t
18 // generic pack/unpack routines.
19 //
20 // TODO(rsc): There are enough names defined in this file t
21 // prefixed with dns. Perhaps put this in its own package l
22
23 package net
24
25 // Packet formats
26
27 // Wire constants.
28 const (
29     // valid dnsRR_Header.Rrtype and dnsQuestion.qtype
30     dnsTypeA      = 1
31     dnsTypeNS     = 2
32     dnsTypeMD     = 3
33     dnsTypeMF     = 4
34     dnsTypeCNAME  = 5
35     dnsTypeSOA    = 6
36     dnsTypeMB     = 7
37     dnsTypeMG     = 8
38     dnsTypeMR     = 9
39     dnsTypeNULL   = 10
40     dnsTypeWKS    = 11
41     dnsTypePTR    = 12
42     dnsTypeHINFO  = 13
43     dnsTypeMINFO  = 14
44     dnsTypeMX     = 15
```

```

45         dnsTypeTXT      = 16
46         dnsTypeAAAA    = 28
47         dnsTypeSRV     = 33
48
49         // valid dnsQuestion.qtype only
50         dnsTypeAXFR    = 252
51         dnsTypeMAILB   = 253
52         dnsTypeMAILA   = 254
53         dnsTypeALL     = 255
54
55         // valid dnsQuestion.qclass
56         dnsClassINET   = 1
57         dnsClassCSNET  = 2
58         dnsClassCHAOS  = 3
59         dnsClassHESIOD = 4
60         dnsClassANY    = 255
61
62         // dnsMsg.rcode
63         dnsRcodeSuccess      = 0
64         dnsRcodeFormatError  = 1
65         dnsRcodeServerFailure = 2
66         dnsRcodeNameError    = 3
67         dnsRcodeNotImplemented = 4
68         dnsRcodeRefused      = 5
69     )
70
71     // A dnsStruct describes how to iterate over its fields to e
72     // reflective marshalling.
73     type dnsStruct interface {
74         // Walk iterates over fields of a structure and call
75         // with a reference to that field, the name of the f
76         // and a tag ("", "domain", "ipv4", "ipv6") specifyi
77         // particular encodings. Possible concrete types
78         // for v are *uint16, *uint32, *string, or []byte, a
79         // *int, *bool in the case of dnsMsgHdr.
80         // Whenever f returns false, Walk must stop and retu
81         // false, and otherwise return true.
82         Walk(f func(v interface{}, name, tag string) (ok boo
83     }
84
85     // The wire format for the DNS packet header.
86     type dnsHeader struct {
87         Id            uint16
88         Bits          uint16
89         Qdcount, Ancount, Nscount, Arcount uint16
90     }
91
92     func (h *dnsHeader) Walk(f func(v interface{}, name, tag str
93         return f(&h.Id, "Id", "") &&
94             f(&h.Bits, "Bits", "") &&

```

```

95         f(&h.Qdcount, "Qdcount", "") &&
96         f(&h.Ancount, "Ancount", "") &&
97         f(&h.Nscount, "Nscount", "") &&
98         f(&h.Arcount, "Arcount", "")
99     }
100
101     const (
102         // dnsHeader.Bits
103         _QR = 1 << 15 // query/response (response=1)
104         _AA = 1 << 10 // authoritative
105         _TC = 1 << 9  // truncated
106         _RD = 1 << 8  // recursion desired
107         _RA = 1 << 7  // recursion available
108     )
109
110     // DNS queries.
111     type dnsQuestion struct {
112         Name    string `net:"domain-name"` // `net:"domain-na
113         Qtype   uint16
114         Qclass  uint16
115     }
116
117     func (q *dnsQuestion) Walk(f func(v interface{}), name, tag s
118         return f(&q.Name, "Name", "domain") &&
119             f(&q.Qtype, "Qtype", "") &&
120             f(&q.Qclass, "Qclass", "")
121     }
122
123     // DNS responses (resource records).
124     // There are many types of messages,
125     // but they all share the same header.
126     type dnsRR_Header struct {
127         Name    string `net:"domain-name"`
128         Rrtype  uint16
129         Class   uint16
130         Ttl     uint32
131         Rdlength uint16 // length of data after header
132     }
133
134     func (h *dnsRR_Header) Header() *dnsRR_Header {
135         return h
136     }
137
138     func (h *dnsRR_Header) Walk(f func(v interface{}), name, tag
139         return f(&h.Name, "Name", "domain") &&
140             f(&h.Rrtype, "Rrtype", "") &&
141             f(&h.Class, "Class", "") &&
142             f(&h.Ttl, "Ttl", "") &&
143             f(&h.Rdlength, "Rdlength", "")

```

```

144 }
145
146 type dnsRR interface {
147     dnsStruct
148     Header() *dnsRR_Header
149 }
150
151 // Specific DNS RR formats for each query type.
152
153 type dnsRR_CNAME struct {
154     Hdr dnsRR_Header
155     Cname string `net:"domain-name"`
156 }
157
158 func (rr *dnsRR_CNAME) Header() *dnsRR_Header {
159     return &rr.Hdr
160 }
161
162 func (rr *dnsRR_CNAME) Walk(f func(v interface{}), name, tag
163     return rr.Hdr.Walk(f) && f(&rr.Cname, "Cname", "doma
164 }
165
166 type dnsRR_HINFO struct {
167     Hdr dnsRR_Header
168     Cpu string
169     Os string
170 }
171
172 func (rr *dnsRR_HINFO) Header() *dnsRR_Header {
173     return &rr.Hdr
174 }
175
176 func (rr *dnsRR_HINFO) Walk(f func(v interface{}), name, tag
177     return rr.Hdr.Walk(f) && f(&rr.Cpu, "Cpu", "") && f(
178 }
179
180 type dnsRR_MB struct {
181     Hdr dnsRR_Header
182     Mb string `net:"domain-name"`
183 }
184
185 func (rr *dnsRR_MB) Header() *dnsRR_Header {
186     return &rr.Hdr
187 }
188
189 func (rr *dnsRR_MB) Walk(f func(v interface{}), name, tag str
190     return rr.Hdr.Walk(f) && f(&rr.Mb, "Mb", "domain")
191 }
192

```

```

193 type dnsRR_MG struct {
194     Hdr dnsRR_Header
195     Mg string `net:"domain-name"`
196 }
197
198 func (rr *dnsRR_MG) Header() *dnsRR_Header {
199     return &rr.Hdr
200 }
201
202 func (rr *dnsRR_MG) Walk(f func(v interface{}), name, tag str
203     return rr.Hdr.Walk(f) && f(&rr.Mg, "Mg", "domain")
204 }
205
206 type dnsRR_MINFO struct {
207     Hdr dnsRR_Header
208     Rmail string `net:"domain-name"`
209     Email string `net:"domain-name"`
210 }
211
212 func (rr *dnsRR_MINFO) Header() *dnsRR_Header {
213     return &rr.Hdr
214 }
215
216 func (rr *dnsRR_MINFO) Walk(f func(v interface{}), name, tag
217     return rr.Hdr.Walk(f) && f(&rr.Rmail, "Rmail", "doma
218 }
219
220 type dnsRR_MR struct {
221     Hdr dnsRR_Header
222     Mr string `net:"domain-name"`
223 }
224
225 func (rr *dnsRR_MR) Header() *dnsRR_Header {
226     return &rr.Hdr
227 }
228
229 func (rr *dnsRR_MR) Walk(f func(v interface{}), name, tag str
230     return rr.Hdr.Walk(f) && f(&rr.Mr, "Mr", "domain")
231 }
232
233 type dnsRR_MX struct {
234     Hdr dnsRR_Header
235     Pref uint16
236     Mx string `net:"domain-name"`
237 }
238
239 func (rr *dnsRR_MX) Header() *dnsRR_Header {
240     return &rr.Hdr
241 }
242

```

```

243 func (rr *dnsRR_MX) Walk(f func(v interface{}), name, tag str
244     return rr.Hdr.Walk(f) && f(&rr.Pref, "Pref", "") &&
245 }
246
247 type dnsRR_NS struct {
248     Hdr dnsRR_Header
249     Ns string `net:"domain-name"`
250 }
251
252 func (rr *dnsRR_NS) Header() *dnsRR_Header {
253     return &rr.Hdr
254 }
255
256 func (rr *dnsRR_NS) Walk(f func(v interface{}), name, tag str
257     return rr.Hdr.Walk(f) && f(&rr.Ns, "Ns", "domain")
258 }
259
260 type dnsRR_PTR struct {
261     Hdr dnsRR_Header
262     Ptr string `net:"domain-name"`
263 }
264
265 func (rr *dnsRR_PTR) Header() *dnsRR_Header {
266     return &rr.Hdr
267 }
268
269 func (rr *dnsRR_PTR) Walk(f func(v interface{}), name, tag st
270     return rr.Hdr.Walk(f) && f(&rr.Ptr, "Ptr", "domain")
271 }
272
273 type dnsRR_SOA struct {
274     Hdr dnsRR_Header
275     Ns string `net:"domain-name"`
276     Mbox string `net:"domain-name"`
277     Serial uint32
278     Refresh uint32
279     Retry uint32
280     Expire uint32
281     Minttl uint32
282 }
283
284 func (rr *dnsRR_SOA) Header() *dnsRR_Header {
285     return &rr.Hdr
286 }
287
288 func (rr *dnsRR_SOA) Walk(f func(v interface{}), name, tag st
289     return rr.Hdr.Walk(f) &&
290         f(&rr.Ns, "Ns", "domain") &&
291         f(&rr.Mbox, "Mbox", "domain") &&

```

```

292         f(&rr.Serial, "Serial", "") &&
293         f(&rr.Refresh, "Refresh", "") &&
294         f(&rr.Retry, "Retry", "") &&
295         f(&rr.Expire, "Expire", "") &&
296         f(&rr.Minttl, "Minttl", "")
297     }
298
299     type dnsRR_TXT struct {
300         Hdr dnsRR_Header
301         Txt string // not domain name
302     }
303
304     func (rr *dnsRR_TXT) Header() *dnsRR_Header {
305         return &rr.Hdr
306     }
307
308     func (rr *dnsRR_TXT) Walk(f func(v interface{}), name, tag st
309         return rr.Hdr.Walk(f) && f(&rr.Txt, "Txt", "")
310     }
311
312     type dnsRR_SRV struct {
313         Hdr dnsRR_Header
314         Priority uint16
315         Weight uint16
316         Port uint16
317         Target string `net:"domain-name"`
318     }
319
320     func (rr *dnsRR_SRV) Header() *dnsRR_Header {
321         return &rr.Hdr
322     }
323
324     func (rr *dnsRR_SRV) Walk(f func(v interface{}), name, tag st
325         return rr.Hdr.Walk(f) &&
326             f(&rr.Priority, "Priority", "") &&
327             f(&rr.Weight, "Weight", "") &&
328             f(&rr.Port, "Port", "") &&
329             f(&rr.Target, "Target", "domain")
330     }
331
332     type dnsRR_A struct {
333         Hdr dnsRR_Header
334         A uint32 `net:"ipv4"`
335     }
336
337     func (rr *dnsRR_A) Header() *dnsRR_Header {
338         return &rr.Hdr
339     }
340

```

```

341 func (rr *dnsRR_A) Walk(f func(v interface{}), name, tag stri
342     return rr.Hdr.Walk(f) && f(&rr.A, "A", "ipv4")
343 }
344
345 type dnsRR_AAAA struct {
346     Hdr dnsRR_Header
347     AAAA [16]byte `net:"ipv6"`
348 }
349
350 func (rr *dnsRR_AAAA) Header() *dnsRR_Header {
351     return &rr.Hdr
352 }
353
354 func (rr *dnsRR_AAAA) Walk(f func(v interface{}), name, tag s
355     return rr.Hdr.Walk(f) && f(rr.AAAA[:], "AAAA", "ipv6
356 }
357
358 // Packing and unpacking.
359 //
360 // All the packers and unpackers take a (msg []byte, off int
361 // and return (off1 int, ok bool). If they return ok==false
362 // also return off1==len(msg), so that the next unpacker wil
363 // also fail. This lets us avoid checks of ok until the end
364 // packing sequence.
365
366 // Map of constructors for each RR wire type.
367 var rr_mk = map[int]func() dnsRR{
368     dnsTypeCNAME: func() dnsRR { return new(dnsRR_CNAME)
369     dnsTypeHINFO: func() dnsRR { return new(dnsRR_HINFO)
370     dnsTypeMB:     func() dnsRR { return new(dnsRR_MB) },
371     dnsTypeMG:     func() dnsRR { return new(dnsRR_MG) },
372     dnsTypeMINFO: func() dnsRR { return new(dnsRR_MINFO)
373     dnsTypeMR:     func() dnsRR { return new(dnsRR_MR) },
374     dnsTypeMX:     func() dnsRR { return new(dnsRR_MX) },
375     dnsTypeNS:     func() dnsRR { return new(dnsRR_NS) },
376     dnsTypePTR:    func() dnsRR { return new(dnsRR_PTR) }
377     dnsTypeSOA:    func() dnsRR { return new(dnsRR_SOA) }
378     dnsTypeTXT:    func() dnsRR { return new(dnsRR_TXT) }
379     dnsTypeSRV:    func() dnsRR { return new(dnsRR_SRV) }
380     dnsTypeA:      func() dnsRR { return new(dnsRR_A) },
381     dnsTypeAAAA:  func() dnsRR { return new(dnsRR_AAAA)
382 }
383
384 // Pack a domain name s into msg[off:].
385 // Domain names are a sequence of counted strings
386 // split at the dots. They end with a zero-length string.
387 func packDomainName(s string, msg []byte, off int) (off1 int
388     // Add trailing dot to canonicalize name.
389     if n := len(s); n == 0 || s[n-1] != '.' {
390         s += "."

```

```

391     }
392
393     // Each dot ends a segment of the name.
394     // We trade each dot byte for a length byte.
395     // There is also a trailing zero.
396     // Check that we have all the space we need.
397     tot := len(s) + 1
398     if off+tot > len(msg) {
399         return len(msg), false
400     }
401
402     // Emit sequence of counted strings, chopping at dot
403     begin := 0
404     for i := 0; i < len(s); i++ {
405         if s[i] == '.' {
406             if i-begin >= 1<<6 { // top two bits
407                 return len(msg), false
408             }
409             msg[off] = byte(i - begin)
410             off++
411             for j := begin; j < i; j++ {
412                 msg[off] = s[j]
413                 off++
414             }
415             begin = i + 1
416         }
417     }
418     msg[off] = 0
419     off++
420     return off, true
421 }
422
423 // Unpack a domain name.
424 // In addition to the simple sequences of counted strings ab
425 // domain names are allowed to refer to strings elsewhere in
426 // packet, to avoid repeating common suffixes when returning
427 // many entries in a single domain. The pointers are marked
428 // by a length byte with the top two bits set. Ignoring the
429 // two bits, that byte and the next give a 14 bit offset fro
430 // where we should pick up the trail.
431 // Note that if we jump elsewhere in the packet,
432 // we return off1 == the offset after the first pointer we f
433 // which is where the next record will start.
434 // In theory, the pointers are only allowed to jump backward
435 // We let them jump anywhere and stop jumping after a while.
436 func unpackDomainName(msg []byte, off int) (s string, off1 i
437     s = ""
438     ptr := 0 // number of pointers followed
439 Loop:

```

```

440     for {
441         if off >= len(msg) {
442             return "", len(msg), false
443         }
444         c := int(msg[off])
445         off++
446         switch c & 0xC0 {
447         case 0x00:
448             if c == 0x00 {
449                 // end of name
450                 break Loop
451             }
452             // literal string
453             if off+c > len(msg) {
454                 return "", len(msg), false
455             }
456             s += string(msg[off:off+c]) + "."
457             off += c
458         case 0xC0:
459             // pointer to somewhere else in msg.
460             // remember location after first ptr
461             // since that's how many bytes we co
462             // also, don't follow too many point
463             // maybe there's a loop.
464             if off >= len(msg) {
465                 return "", len(msg), false
466             }
467             c1 := msg[off]
468             off++
469             if ptr == 0 {
470                 off1 = off
471             }
472             if ptr++; ptr > 10 {
473                 return "", len(msg), false
474             }
475             off = (c^0xC0)<<8 | int(c1)
476         default:
477             // 0x80 and 0x40 are reserved
478             return "", len(msg), false
479         }
480     }
481     if ptr == 0 {
482         off1 = off
483     }
484     return s, off1, true
485 }
486
487 // packStruct packs a structure into msg at specified offset
488 // returns off1 such that msg[off:off1] is the encoded data.

```

```

489 func packStruct(any dnsStruct, msg []byte, off int) (off1 in
490     ok = any.Walk(func(field interface{}), name, tag stri
491     switch fv := field.(type) {
492     default:
493         println("net: dns: unknown packing t
494         return false
495     case *uint16:
496         i := *fv
497         if off+2 > len(msg) {
498             return false
499         }
500         msg[off] = byte(i >> 8)
501         msg[off+1] = byte(i)
502         off += 2
503     case *uint32:
504         i := *fv
505         msg[off] = byte(i >> 24)
506         msg[off+1] = byte(i >> 16)
507         msg[off+2] = byte(i >> 8)
508         msg[off+3] = byte(i)
509         off += 4
510     case []byte:
511         n := len(fv)
512         if off+n > len(msg) {
513             return false
514         }
515         copy(msg[off:off+n], fv)
516         off += n
517     case *string:
518         s := *fv
519         switch tag {
520         default:
521             println("net: dns: unknown s
522             return false
523         case "domain":
524             off, ok = packDomainName(s,
525             if !ok {
526                 return false
527             }
528         case "":
529             // Counted string: 1 byte le
530             if len(s) > 255 || off+1+len
531                 return false
532             }
533             msg[off] = byte(len(s))
534             off++
535             off += copy(msg[off:], s)
536         }
537     }
538     return true

```

```

539         })
540         if !ok {
541             return len(msg), false
542         }
543         return off, true
544     }
545
546     // unpackStruct decodes msg[off:] into the given structure,
547     // returns off1 such that msg[off:off1] is the encoded data.
548     func unpackStruct(any dnsStruct, msg []byte, off int) (off1
549         ok = any.Walk(func(field interface{}, name, tag stri
550             switch fv := field.(type) {
551             default:
552                 println("net: dns: unknown packing t
553                 return false
554             case *uint16:
555                 if off+2 > len(msg) {
556                     return false
557                 }
558                 *fv = uint16(msg[off])<<8 | uint16(m
559                 off += 2
560             case *uint32:
561                 if off+4 > len(msg) {
562                     return false
563                 }
564                 *fv = uint32(msg[off])<<24 | uint32(
565                     uint32(msg[off+2])<<8 | uint
566                 off += 4
567             case []byte:
568                 n := len(fv)
569                 if off+n > len(msg) {
570                     return false
571                 }
572                 copy(fv, msg[off:off+n])
573                 off += n
574             case *string:
575                 var s string
576                 switch tag {
577                 default:
578                     println("net: dns: unknown s
579                     return false
580                 case "domain":
581                     s, off, ok = unpackDomainNar
582                     if !ok {
583                         return false
584                     }
585                 case "":
586                     if off >= len(msg) || off+1+
587                         return false

```

```

588         }
589         n := int(msg[off])
590         off++
591         b := make([]byte, n)
592         for i := 0; i < n; i++ {
593             b[i] = msg[off+i]
594         }
595         off += n
596         s = string(b)
597     }
598     *fv = s
599 }
600 return true
601 })
602 if !ok {
603     return len(msg), false
604 }
605 return off, true
606 }
607
608 // Generic struct printer. Prints fields with tag "ipv4" or
609 // as IP addresses.
610 func printStruct(any dnsStruct) string {
611     s := "{"
612     i := 0
613     any.Walk(func(val interface{}, name, tag string) bool {
614         i++
615         if i > 1 {
616             s += ", "
617         }
618         s += name + "="
619         switch tag {
620         case "ipv4":
621             i := val.(uint32)
622             s += IPv4(byte(i>>24), byte(i>>16)),
623         case "ipv6":
624             i := val.([]byte)
625             s += IP(i).String()
626         default:
627             var i int64
628             switch v := val.(type) {
629             default:
630                 // can't really happen.
631                 s += "<unknown type>"
632                 return true
633             case *string:
634                 s += *v
635                 return true
636             case []byte:

```

```

637         s += string(v)
638         return true
639     case *bool:
640         if *v {
641             s += "true"
642         } else {
643             s += "false"
644         }
645         return true
646     case *int:
647         i = int64(*v)
648     case *uint:
649         i = int64(*v)
650     case *uint8:
651         i = int64(*v)
652     case *uint16:
653         i = int64(*v)
654     case *uint32:
655         i = int64(*v)
656     case *uint64:
657         i = int64(*v)
658     case *uintptr:
659         i = int64(*v)
660     }
661     s += itoa(int(i))
662 }
663 return true
664 })
665 s += "}"
666 return s
667 }
668
669 // Resource record packer.
670 func packRR(rr dnsRR, msg []byte, off int) (off2 int, ok boo
671     var off1 int
672     // pack twice, once to find end of header
673     // and again to find end of packet.
674     // a bit inefficient but this doesn't need to be fas
675     // off1 is end of header
676     // off2 is end of rr
677     off1, ok = packStruct(rr.Header(), msg, off)
678     off2, ok = packStruct(rr, msg, off)
679     if !ok {
680         return len(msg), false
681     }
682     // pack a third time; redo header with correct data
683     rr.Header().Rdlength = uint16(off2 - off1)
684     packStruct(rr.Header(), msg, off)
685     return off2, true
686 }

```

```

687
688 // Resource record unpacker.
689 func unpackRR(msg []byte, off int) (rr dnsRR, off1 int, ok bool) {
690     // unpack just the header, to find the rr type and length
691     var h dnsRR_Header
692     off0 := off
693     if off, ok = unpackStruct(&h, msg, off); !ok {
694         return nil, len(msg), false
695     }
696     end := off + int(h.Rdlength)
697
698     // make an rr of that type and re-unpack.
699     // again inefficient but doesn't need to be fast.
700     mk, known := rr_mk[int(h.Rrtype)]
701     if !known {
702         return &h, end, true
703     }
704     rr = mk()
705     off, ok = unpackStruct(rr, msg, off0)
706     if off != end {
707         return &h, end, true
708     }
709     return rr, off, ok
710 }
711
712 // Usable representation of a DNS packet.
713
714 // A manually-unpacked version of (id, bits).
715 // This is in its own struct for easy printing.
716 type dnsMsgHdr struct {
717     id                uint16
718     response          bool
719     opcode            int
720     authoritative    bool
721     truncated         bool
722     recursion_desired bool
723     recursion_available bool
724     rcode             int
725 }
726
727 func (h *dnsMsgHdr) Walk(f func(v interface{}, name, tag string) bool) bool {
728     return f(&h.id, "id", "") &&
729         f(&h.response, "response", "") &&
730         f(&h.opcode, "opcode", "") &&
731         f(&h.authoritative, "authoritative", "") &&
732         f(&h.truncated, "truncated", "") &&
733         f(&h.recursion_desired, "recursion_desired", "") &&
734         f(&h.recursion_available, "recursion_available", "") &&
735         f(&h.rcode, "rcode", "")

```

```

736 }
737
738 type dnsMsg struct {
739     dnsMsgHdr
740     question []dnsQuestion
741     answer    []dnsRR
742     ns        []dnsRR
743     extra     []dnsRR
744 }
745
746 func (dns *dnsMsg) Pack() (msg []byte, ok bool) {
747     var dh dnsHeader
748
749     // Convert convenient dnsMsg into wire-like dnsHeade
750     dh.Id = dns.id
751     dh.Bits = uint16(dns.opcode)<<11 | uint16(dns.rcode)
752     if dns.recursion_available {
753         dh.Bits |= _RA
754     }
755     if dns.recursion_desired {
756         dh.Bits |= _RD
757     }
758     if dns.truncated {
759         dh.Bits |= _TC
760     }
761     if dns.authoritative {
762         dh.Bits |= _AA
763     }
764     if dns.response {
765         dh.Bits |= _QR
766     }
767
768     // Prepare variable sized arrays.
769     question := dns.question
770     answer := dns.answer
771     ns := dns.ns
772     extra := dns.extra
773
774     dh.Qdcount = uint16(len(question))
775     dh.Ancount = uint16(len(answer))
776     dh.Nscount = uint16(len(ns))
777     dh.Arcount = uint16(len(extra))
778
779     // Could work harder to calculate message size,
780     // but this is far more than we need and not
781     // big enough to hurt the allocator.
782     msg = make([]byte, 2000)
783
784     // Pack it in: header and then the pieces.

```

```

785     off := 0
786     off, ok = packStruct(&dh, msg, off)
787     for i := 0; i < len(question); i++ {
788         off, ok = packStruct(&question[i], msg, off)
789     }
790     for i := 0; i < len(answer); i++ {
791         off, ok = packRR(answer[i], msg, off)
792     }
793     for i := 0; i < len(ns); i++ {
794         off, ok = packRR(ns[i], msg, off)
795     }
796     for i := 0; i < len(extra); i++ {
797         off, ok = packRR(extra[i], msg, off)
798     }
799     if !ok {
800         return nil, false
801     }
802     return msg[0:off], true
803 }
804
805 func (dns *dnsMsg) Unpack(msg []byte) bool {
806     // Header.
807     var dh dnsHeader
808     off := 0
809     var ok bool
810     if off, ok = unpackStruct(&dh, msg, off); !ok {
811         return false
812     }
813     dns.id = dh.Id
814     dns.response = (dh.Bits & _QR) != 0
815     dns.opcode = int(dh.Bits >> 11) & 0xF
816     dns.authoritative = (dh.Bits & _AA) != 0
817     dns.truncated = (dh.Bits & _TC) != 0
818     dns.recursion_desired = (dh.Bits & _RD) != 0
819     dns.recursion_available = (dh.Bits & _RA) != 0
820     dns.rcode = int(dh.Bits & 0xF)
821
822     // Arrays.
823     dns.question = make([]dnsQuestion, dh.Qdcount)
824     dns.answer = make([]dnsRR, 0, dh.Ancount)
825     dns.ns = make([]dnsRR, 0, dh.Nscount)
826     dns.extra = make([]dnsRR, 0, dh.Arcount)
827
828     var rec dnsRR
829
830     for i := 0; i < len(dns.question); i++ {
831         off, ok = unpackStruct(&dns.question[i], msg
832     }
833     for i := 0; i < int(dh.Ancount); i++ {
834         rec, off, ok = unpackRR(msg, off)

```

```

835         if !ok {
836             return false
837         }
838         dns.answer = append(dns.answer, rec)
839     }
840     for i := 0; i < int(dh.Nscount); i++ {
841         rec, off, ok = unpackRR(msg, off)
842         if !ok {
843             return false
844         }
845         dns.ns = append(dns.ns, rec)
846     }
847     for i := 0; i < int(dh.Arcount); i++ {
848         rec, off, ok = unpackRR(msg, off)
849         if !ok {
850             return false
851         }
852         dns.extra = append(dns.extra, rec)
853     }
854     // if off != len(msg) {
855     //     println("extra bytes in dns packet",
856     //         }
857     return true
858 }
859
860 func (dns *dnsMsg) String() string {
861     s := "DNS: " + printStruct(&dns.dnsMsgHdr) + "\n"
862     if len(dns.question) > 0 {
863         s += "-- Questions\n"
864         for i := 0; i < len(dns.question); i++ {
865             s += printStruct(&dns.question[i]) +
866         }
867     }
868     if len(dns.answer) > 0 {
869         s += "-- Answers\n"
870         for i := 0; i < len(dns.answer); i++ {
871             s += printStruct(dns.answer[i]) + "\
872         }
873     }
874     if len(dns.ns) > 0 {
875         s += "-- Name servers\n"
876         for i := 0; i < len(dns.ns); i++ {
877             s += printStruct(dns.ns[i]) + "\n"
878         }
879     }
880     if len(dns.extra) > 0 {
881         s += "-- Extra\n"
882         for i := 0; i < len(dns.extra); i++ {
883             s += printStruct(dns.extra[i]) + "\n

```

```
884         }
885     }
886     return s
887 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/doc.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package net
6
7 // LookupHost looks up the given host using the local resolver.
8 // It returns an array of that host's addresses.
9 func LookupHost(host string) (addrs []string, err error) {
10     return lookupHost(host)
11 }
12
13 // LookupIP looks up host using the local resolver.
14 // It returns an array of that host's IPv4 and IPv6 addresses.
15 func LookupIP(host string) (addrs []IP, err error) {
16     return lookupIP(host)
17 }
18
19 // LookupPort looks up the port for the given network and service.
20 func LookupPort(network, service string) (port int, err error) {
21     return lookupPort(network, service)
22 }
23
24 // LookupCNAME returns the canonical DNS host for the given
25 // host. Callers that do not care about the canonical name can call
26 // LookupHost or LookupIP directly; both take care of resolving
27 // the canonical name as part of the lookup.
28 func LookupCNAME(name string) (cname string, err error) {
29     return lookupCNAME(name)
30 }
31
32 // LookupSRV tries to resolve an SRV query of the given service,
33 // protocol, and domain name. The proto is "tcp" or "udp".
34 // The returned records are sorted by priority and randomized
35 // by weight within a priority.
36 //
37 // LookupSRV constructs the DNS name to look up following RFC
38 // 2782. That is, it looks up _service._proto.name. To accommodate
39 // publishing SRV records under non-standard names, if both
40 // service and proto are empty strings, LookupSRV looks up name directly.
41 func LookupSRV(service, proto, name string) (cname string, addrs []IP, err error) {
42     return lookupSRV(service, proto, name)
43 }
44
```

```
45 // LookupMX returns the DNS MX records for the given domain
46 func LookupMX(name string) (mx []*MX, err error) {
47     return lookupMX(name)
48 }
49
50 // LookupTXT returns the DNS TXT records for the given domain
51 func LookupTXT(name string) (txt []string, err error) {
52     return lookupTXT(name)
53 }
54
55 // LookupAddr performs a reverse lookup for the given address
56 // of names mapping to that address.
57 func LookupAddr(addr string) (name []string, err error) {
58     return lookupAddr(addr)
59 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/fd.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package net
8
9 import (
10     "errors"
11     "io"
12     "os"
13     "sync"
14     "syscall"
15     "time"
16 )
17
18 // Network file descriptor.
19 type netFD struct {
20     // locking/lifetime of sysfd
21     sysmu sync.Mutex
22     sysref int
23
24     // must lock both sysmu and pollserver to write
25     // can lock either to read
26     closing bool
27
28     // immutable until Close
29     sysfd int
30     family int
31     sotype int
32     isConnected bool
33     sysfile *os.File
34     cr chan error
35     cw chan error
36     net string
37     laddr Addr
38     raddr Addr
39
40     // owned by client
41     rdeadline int64
42     rio sync.Mutex
43     wdeadline int64
44     wio sync.Mutex
```

```

45
46         // owned by fd wait server
47         ncr, ncw int
48     }
49
50 // A pollServer helps FDs determine when to retry a non-bloc
51 // read or write after they get EAGAIN. When an FD needs to
52 // send the fd on s.cr (for a read) or s.cw (for a write) to
53 // request to the poll server. Then receive on fd.cr/fd.cw.
54 // When the pollServer finds that i/o on FD should be possib
55 // again, it will send fd on fd.cr/fd.cw to wake any waiting
56 // This protocol is implemented as s.WaitRead() and s.WaitWr
57 //
58 // There is one subtlety: when sending on s.cr/s.cw, the
59 // poll server is probably in a system call, waiting for an
60 // to become ready. It's not looking at the request channel
61 // To resolve this, the poll server waits not just on the FD
62 // been given but also its own pipe. After sending on the
63 // buffered channel s.cr/s.cw, WaitRead/WaitWrite writes a
64 // byte to the pipe, causing the pollServer's poll system ca
65 // return. In response to the pipe being readable, the poll
66 // re-polls its request channels.
67 //
68 // Note that the ordering is "send request" and then "wake u
69 // If the operations were reversed, there would be a race: t
70 // server might wake up and look at the request channel, see
71 // was empty, and go back to sleep, all before the requester
72 // to send the request. Because the send must complete befo
73 // the request channel must be buffered. A buffer of size 1
74 // for any request load. If many processes are trying to su
75 // one will succeed, the pollServer will read the request, a
76 // channel will be empty for the next process's request. A
77 // might help batch requests.
78 //
79 // To avoid races in closing, all fd operations are locked a
80 // refcounted. when netFD.Close() is called, it calls syscal
81 // and sets a closing flag. Only when the last reference is
82 // will the fd be closed.
83
84 type pollServer struct {
85     cr, cw     chan *netFD // buffered >= 1
86     pr, pw     *os.File
87     poll       *pollster // low-level OS hooks
88     sync.Mutex // controls pending and deadlin
89     pending    map[int]*netFD
90     deadline   int64 // next deadline (nsec since 1970)
91 }
92
93 func (s *pollServer) AddFD(fd *netFD, mode int) error {
94     s.Lock()

```

```

95         intfd := fd.sysfd
96         if intfd < 0 || fd.closing {
97             // fd closed underfoot
98             s.Unlock()
99             return errClosing
100        }
101
102        var t int64
103        key := intfd << 1
104        if mode == 'r' {
105            fd.ncr++
106            t = fd.rdeadline
107        } else {
108            fd.ncw++
109            key++
110            t = fd.wdeadline
111        }
112        s.pending[key] = fd
113        doWakeup := false
114        if t > 0 && (s.deadline == 0 || t < s.deadline) {
115            s.deadline = t
116            doWakeup = true
117        }
118
119        wake, err := s.poll.AddFD(intfd, mode, false)
120        if err != nil {
121            panic("pollServer AddFD " + err.Error())
122        }
123        if wake {
124            doWakeup = true
125        }
126        s.Unlock()
127
128        if doWakeup {
129            s.Wakeup()
130        }
131        return nil
132    }
133
134    // Evict evicts fd from the pending list, unblocking
135    // any I/O running on fd. The caller must have locked
136    // pollserver.
137    func (s *pollServer) Evict(fd *netFD) {
138        if s.pending[fd.sysfd<<1] == fd {
139            s.WakeFD(fd, 'r', errClosing)
140            s.poll.DelFD(fd.sysfd, 'r')
141            delete(s.pending, fd.sysfd<<1)
142        }
143        if s.pending[fd.sysfd<<1|1] == fd {

```

```

144             s.WakeFD(fd, 'w', errClosing)
145             s.poll.DelFD(fd.sysfd, 'w')
146             delete(s.pending, fd.sysfd<<1|1)
147         }
148     }
149
150     var wakeupbuf [1]byte
151
152     func (s *pollServer) Wakeup() { s.pw.Write(wakeupbuf[0:]) }
153
154     func (s *pollServer) LookupFD(fd int, mode int) *netFD {
155         key := fd << 1
156         if mode == 'w' {
157             key++
158         }
159         netfd, ok := s.pending[key]
160         if !ok {
161             return nil
162         }
163         delete(s.pending, key)
164         return netfd
165     }
166
167     func (s *pollServer) WakeFD(fd *netFD, mode int, err error)
168         if mode == 'r' {
169             for fd.ncr > 0 {
170                 fd.ncr--
171                 fd.cr <- err
172             }
173         } else {
174             for fd.ncw > 0 {
175                 fd.ncw--
176                 fd.cw <- err
177             }
178         }
179     }
180
181     func (s *pollServer) Now() int64 {
182         return time.Now().UnixNano()
183     }
184
185     func (s *pollServer) CheckDeadlines() {
186         now := s.Now()
187         // TODO(rsc): This will need to be handled more effi
188         // probably with a heap indexed by wakeup time.
189
190         var next_deadline int64
191         for key, fd := range s.pending {
192             var t int64

```

```

193         var mode int
194         if key&1 == 0 {
195             mode = 'r'
196         } else {
197             mode = 'w'
198         }
199         if mode == 'r' {
200             t = fd.rdeadline
201         } else {
202             t = fd.wdeadline
203         }
204         if t > 0 {
205             if t <= now {
206                 delete(s.pending, key)
207                 if mode == 'r' {
208                     s.poll.DelFD(fd.sysf
209                         fd.rdeadline = -1
210                 } else {
211                     s.poll.DelFD(fd.sysf
212                         fd.wdeadline = -1
213                 }
214                 s.WakeFD(fd, mode, nil)
215             } else if next_deadline == 0 || t <
216                 next_deadline = t
217             }
218         }
219     }
220     s.deadline = next_deadline
221 }
222
223 func (s *pollServer) Run() {
224     var scratch [100]byte
225     s.Lock()
226     defer s.Unlock()
227     for {
228         var t = s.deadline
229         if t > 0 {
230             t = t - s.Now()
231             if t <= 0 {
232                 s.CheckDeadlines()
233                 continue
234             }
235         }
236         fd, mode, err := s.poll.WaitFD(s, t)
237         if err != nil {
238             print("pollServer WaitFD: ", err.Err
239             return
240         }
241         if fd < 0 {
242             // Timeout happened.

```

```

243             s.CheckDeadlines()
244             continue
245         }
246         if fd == int(s.pr.Fd()) {
247             // Drain our wakeup pipe (we could l
248             // but it's unlikely that there are
249             // len(scratch) wakeup calls).
250             s.pr.Read(scratch[0:])
251             s.CheckDeadlines()
252         } else {
253             netfd := s.LookupFD(fd, mode)
254             if netfd == nil {
255                 // This can happen because t
256                 // holding s's lock, so ther
257                 // for an fd that has been e
258                 continue
259             }
260             s.WakeFD(netfd, mode, nil)
261         }
262     }
263 }
264
265 func (s *pollServer) WaitRead(fd *netFD) error {
266     err := s.AddFD(fd, 'r')
267     if err == nil {
268         err = <-fd.cr
269     }
270     return err
271 }
272
273 func (s *pollServer) WaitWrite(fd *netFD) error {
274     err := s.AddFD(fd, 'w')
275     if err == nil {
276         err = <-fd.cw
277     }
278     return err
279 }
280
281 // Network FD methods.
282 // All the network FDs use a single pollServer.
283
284 var pollserver *pollServer
285 var onceStartServer sync.Once
286
287 func startServer() {
288     p, err := newPollServer()
289     if err != nil {
290         print("Start pollServer: ", err.Error(), "\n")
291     }

```

```

292     pollserver = p
293 }
294
295 func newFD(fd, family, sotype int, net string) (*netFD, error)
296     onceStartServer.Do(startServer)
297     if err := syscall.SetNonblock(fd, true); err != nil
298         return nil, err
299     }
300     netfd := &netFD{
301         sysfd:  fd,
302         family: family,
303         sotype: sotype,
304         net:    net,
305     }
306     netfd.cr = make(chan error, 1)
307     netfd.cw = make(chan error, 1)
308     return netfd, nil
309 }
310
311 func (fd *netFD) setAddr(laddr, raddr Addr) {
312     fd.laddr = laddr
313     fd.raddr = raddr
314     var ls, rs string
315     if laddr != nil {
316         ls = laddr.String()
317     }
318     if raddr != nil {
319         rs = raddr.String()
320     }
321     fd.sysfile = os.NewFile(uintptr(fd.sysfd), fd.net+":
322 }
323
324 func (fd *netFD) connect(ra syscall.Sockaddr) error {
325     err := syscall.Connect(fd.sysfd, ra)
326     if err == syscall.EINPROGRESS {
327         if err = pollserver.WaitWrite(fd); err != nil
328             return err
329         }
330         var e int
331         e, err = syscall.GetsockoptInt(fd.sysfd, sys
332         if err != nil {
333             return os.NewSyscallError("getsockoptop
334         }
335         if e != 0 {
336             err = syscall.Errno(e)
337         }
338     }
339     return err
340 }

```

```

341
342 var errClosing = errors.New("use of closed network connectio
343
344 // Add a reference to this fd.
345 // If closing==true, pollserver must be locked; mark the fd
346 // Returns an error if the fd cannot be used.
347 func (fd *netFD) incref(closing bool) error {
348     if fd == nil {
349         return errClosing
350     }
351     fd.sysmu.Lock()
352     if fd.closing {
353         fd.sysmu.Unlock()
354         return errClosing
355     }
356     fd.sysref++
357     if closing {
358         fd.closing = true
359     }
360     fd.sysmu.Unlock()
361     return nil
362 }
363
364 // Remove a reference to this FD and close if we've been ask
365 // there are no references left.
366 func (fd *netFD) decref() {
367     if fd == nil {
368         return
369     }
370     fd.sysmu.Lock()
371     fd.sysref--
372     if fd.closing && fd.sysref == 0 && fd.sysfile != nil
373         fd.sysfile.Close()
374         fd.sysfile = nil
375         fd.sysfd = -1
376     }
377     fd.sysmu.Unlock()
378 }
379
380 func (fd *netFD) Close() error {
381     pollserver.Lock() // needed for both fd.incref(true)
382     defer pollserver.Unlock()
383     if err := fd.incref(true); err != nil {
384         return err
385     }
386     // Unblock any I/O. Once it all unblocks and return
387     // so that it cannot be referring to fd.sysfd anymore
388     // the final decref will close fd.sysfd. This should
389     // fairly quickly, since all the I/O is non-blocking
390     // attempts to block in the pollserver will return e

```

```

391         pollserver.Evict(fd)
392         fd.decref()
393         return nil
394     }
395
396     func (fd *netFD) shutdown(how int) error {
397         if err := fd.incref(false); err != nil {
398             return err
399         }
400         defer fd.decref()
401         err := syscall.Shutdown(fd.sysfd, how)
402         if err != nil {
403             return &OpError{"shutdown", fd.net, fd.laddr
404         }
405         return nil
406     }
407
408     func (fd *netFD) CloseRead() error {
409         return fd.shutdown(syscall.SHUT_RD)
410     }
411
412     func (fd *netFD) CloseWrite() error {
413         return fd.shutdown(syscall.SHUT_WR)
414     }
415
416     func (fd *netFD) Read(p []byte) (n int, err error) {
417         fd.rio.Lock()
418         defer fd.rio.Unlock()
419         if err := fd.incref(false); err != nil {
420             return 0, err
421         }
422         defer fd.decref()
423         for {
424             n, err = syscall.Read(int(fd.sysfd), p)
425             if err == syscall.EAGAIN {
426                 err = errTimeout
427                 if fd.rdeadline >= 0 {
428                     if err = pollserver.WaitRead
429                         continue
430                 }
431             }
432         }
433         if err != nil {
434             n = 0
435         } else if n == 0 && err == nil && fd.sotype
436             err = io.EOF
437         }
438         break
439     }

```



```

489         }
490     }
491     if err == nil && n == 0 {
492         err = io.EOF
493     }
494     break
495 }
496 if err != nil && err != io.EOF {
497     err = &OpError{"read", fd.net, fd.laddr, err}
498     return
499 }
500 return
501 }
502
503 func (fd *netFD) Write(p []byte) (int, error) {
504     fd.wio.Lock()
505     defer fd.wio.Unlock()
506     if err := fd.incref(false); err != nil {
507         return 0, err
508     }
509     defer fd.decref()
510     if fd.sysfile == nil {
511         return 0, syscall.EINVAL
512     }
513
514     var err error
515     nn := 0
516     for {
517         var n int
518         n, err = syscall.Write(int(fd.sysfd), p[nn:])
519         if n > 0 {
520             nn += n
521         }
522         if nn == len(p) {
523             break
524         }
525         if err == syscall.EAGAIN {
526             err = errTimeout
527             if fd.wdeadline >= 0 {
528                 if err = pollserver.WaitWrite(
529                     fd.wdeadline, fd.wio); err != nil {
530                     continue
531                 }
532             }
533         }
534         if err != nil {
535             n = 0
536             break
537         }
538         if n == 0 {
539             err = io.ErrUnexpectedEOF

```

```

539             break
540         }
541     }
542     if err != nil {
543         err = &OpError{"write", fd.net, fd.raddr, er
544     }
545     return nn, err
546 }
547
548 func (fd *netFD) WriteTo(p []byte, sa syscall.Sockaddr) (n i
549     fd.wio.Lock()
550     defer fd.wio.Unlock()
551     if err := fd.incref(false); err != nil {
552         return 0, err
553     }
554     defer fd.decref()
555     for {
556         err = syscall.Sendto(fd.sysfd, p, 0, sa)
557         if err == syscall.EAGAIN {
558             err = errTimeout
559             if fd.wdeadline >= 0 {
560                 if err = pollserver.WaitWrit
561                     continue
562             }
563         }
564     }
565     break
566 }
567 if err == nil {
568     n = len(p)
569 } else {
570     err = &OpError{"write", fd.net, fd.raddr, er
571 }
572 return
573 }
574
575 func (fd *netFD) WriteMsg(p []byte, oob []byte, sa syscall.S
576     fd.wio.Lock()
577     defer fd.wio.Unlock()
578     if err := fd.incref(false); err != nil {
579         return 0, 0, err
580     }
581     defer fd.decref()
582     for {
583         err = syscall.Sendmsg(fd.sysfd, p, oob, sa,
584         if err == syscall.EAGAIN {
585             err = errTimeout
586             if fd.wdeadline >= 0 {
587                 if err = pollserver.WaitWrit

```

```

588                                     continue
589                                     }
590                                 }
591                             }
592                             break
593                         }
594                     if err == nil {
595                         n = len(p)
596                         oobn = len(oob)
597                     } else {
598                         err = &OpError{"write", fd.net, fd.raddr, er
599                     }
600                     return
601 }
602
603 func (fd *netFD) accept(toAddr func(syscall.Sockaddr) Addr)
604     if err := fd.incref(false); err != nil {
605         return nil, err
606     }
607     defer fd.decref()
608
609     // See ../syscall/exec.go for description of ForkLoc
610     // It is okay to hold the lock across syscall.Accept
611     // because we have put fd.sysfd into non-blocking mo
612     var s int
613     var rsa syscall.Sockaddr
614     for {
615         syscall.ForkLock.RLock()
616         s, rsa, err = syscall.Accept(fd.sysfd)
617         if err != nil {
618             syscall.ForkLock.RUnlock()
619             if err == syscall.EAGAIN {
620                 err = errTimeout
621                 if fd.rdeadline >= 0 {
622                     if err = pollserver.
623                         continue
624                 }
625             }
626         } else if err == syscall.ECONNABORTE
627             // This means that a socket
628             // before we Accept()ed it;
629             continue
630         }
631         return nil, &OpError{"accept", fd.ne
632     }
633     break
634 }
635 syscall.CloseOnExec(s)
636 syscall.ForkLock.RUnlock()

```

```

637
638     if netfd, err = newFD(s, fd.family, fd.sotype, fd.ne
639         syscall.Close(s)
640         return nil, err
641     }
642     lsa, _ := syscall.Getsockname(netfd.sysfd)
643     netfd.setAddr(toAddr(lsa), toAddr(rsa))
644     return netfd, nil
645 }
646
647 func (fd *netFD) dup() (f *os.File, err error) {
648     ns, err := syscall.Dup(fd.sysfd)
649     if err != nil {
650         return nil, &OpError{"dup", fd.net, fd.laddr
651     }
652
653     // We want blocking mode for the new fd, hence the d
654     if err = syscall.SetNonblock(ns, false); err != nil
655         return nil, &OpError{"setnonblock", fd.net,
656     }
657
658     return os.NewFile(uintptr(ns), fd.sysfile.Name()), n
659 }
660
661 func closesocket(s int) error {
662     return syscall.Close(s)
663 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/fd_linux.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Waiting for FDs via epoll(7).
6
7 package net
8
9 import (
10     "os"
11     "syscall"
12 )
13
14 const (
15     readFlags = syscall.EPOLLIN | syscall.EPOLLRDHUP
16     writeFlags = syscall.EPOLLOUT
17 )
18
19 type pollster struct {
20     epfd int
21
22     // Events we're already waiting for
23     // Must hold pollServer lock
24     events map[int]uint32
25
26     // An event buffer for EpollWait.
27     // Used without a lock, may only be used by WaitFD.
28     waitEventBuf [10]syscall.EpollEvent
29     waitEvents   []syscall.EpollEvent
30
31     // An event buffer for EpollCtl, to avoid a malloc.
32     // Must hold pollServer lock.
33     ctlEvent syscall.EpollEvent
34 }
35
36 func newpollster() (p *pollster, err error) {
37     p = new(pollster)
38     if p.epfd, err = syscall.EpollCreate1(syscall.EPOLL_
39         if err != syscall.ENOSYS {
40             return nil, os.NewSyscallError("epol
41         }
42         // The arg to epoll_create is a hint to the
43         // about the number of FDs we will care about
44         // We don't know, and since 2.6.8 the kernel
```

```

45         if p.epfd, err = syscall.EpollCreate(16); er
46             return nil, os.NewSyscallError("epol
47         }
48         syscall.CloseOnExec(p.epfd)
49     }
50     p.events = make(map[int]uint32)
51     return p, nil
52 }
53
54 func (p *pollster) AddFD(fd int, mode int, repeat bool) (boo
55     // pollServer is locked.
56
57     var already bool
58     p.ctrlEvent.Fd = int32(fd)
59     p.ctrlEvent.Events, already = p.events[fd]
60     if !repeat {
61         p.ctrlEvent.Events |= syscall.EPOLLONESHOT
62     }
63     if mode == 'r' {
64         p.ctrlEvent.Events |= readFlags
65     } else {
66         p.ctrlEvent.Events |= writeFlags
67     }
68
69     var op int
70     if already {
71         op = syscall.EPOLL_CTL_MOD
72     } else {
73         op = syscall.EPOLL_CTL_ADD
74     }
75     if err := syscall.EpollCtl(p.epfd, op, fd, &p.ctrlEve
76         return false, os.NewSyscallError("epoll_ctl"
77     }
78     p.events[fd] = p.ctrlEvent.Events
79     return false, nil
80 }
81
82 func (p *pollster) StopWaiting(fd int, bits uint) {
83     // pollServer is locked.
84
85     events, already := p.events[fd]
86     if !already {
87         // The fd returned by the kernel may have be
88         // cancelled already; return silently.
89         return
90     }
91
92     // If syscall.EPOLLONESHOT is not set, the wait
93     // is a repeating wait, so don't change it.
94     if events&syscall.EPOLLONESHOT == 0 {

```

```

95         return
96     }
97
98     // Disable the given bits.
99     // If we're still waiting for other events, modify t
100    // event in the kernel. Otherwise, delete it.
101    events &= ^uint32(bits)
102    if int32(events)&^syscall.EPOLLONESHOT != 0 {
103        p.ctrlEvent.Fd = int32(fd)
104        p.ctrlEvent.Events = events
105        if err := syscall.EpollCtl(p.epfd, syscall.E
106            print("Epoll modify fd=", fd, ": ",
107        }
108        p.events[fd] = events
109    } else {
110        if err := syscall.EpollCtl(p.epfd, syscall.E
111            print("Epoll delete fd=", fd, ": ",
112        }
113        delete(p.events, fd)
114    }
115 }
116
117 func (p *pollster) DelFD(fd int, mode int) {
118     // pollServer is locked.
119
120     if mode == 'r' {
121         p.StopWaiting(fd, readFlags)
122     } else {
123         p.StopWaiting(fd, writeFlags)
124     }
125
126     // Discard any queued up events.
127     i := 0
128     for i < len(p.waitEvents) {
129         if fd == int(p.waitEvents[i].Fd) {
130             copy(p.waitEvents[i:], p.waitEvents[
131                 p.waitEvents = p.waitEvents[:len(p.w
132             } else {
133                 i++
134             }
135     }
136 }
137
138 func (p *pollster) WaitFD(s *pollServer, nsec int64) (fd int
139     for len(p.waitEvents) == 0 {
140         var msec int = -1
141         if nsec > 0 {
142             msec = int((nsec + 1e6 - 1) / 1e6)
143         }

```

```

144
145         s.Unlock()
146         n, err := syscall.EpollWait(p.epfd, p.waitEv
147         s.Lock()
148
149         if err != nil {
150             if err == syscall.EAGAIN || err == s
151             continue
152         }
153         return -1, 0, os.NewSyscallError("ep
154     }
155     if n == 0 {
156         return -1, 0, nil
157     }
158     p.waitEvents = p.waitEventBuf[0:n]
159 }
160
161 ev := &p.waitEvents[0]
162 p.waitEvents = p.waitEvents[1:]
163
164 fd = int(ev.Fd)
165
166 if ev.Events&writeFlags != 0 {
167     p.StopWaiting(fd, writeFlags)
168     return fd, 'w', nil
169 }
170 if ev.Events&readFlags != 0 {
171     p.StopWaiting(fd, readFlags)
172     return fd, 'r', nil
173 }
174
175 // Other events are error conditions - wake whoever
176 events, _ := p.events[fd]
177 if events&writeFlags != 0 {
178     p.StopWaiting(fd, writeFlags)
179     return fd, 'w', nil
180 }
181 p.StopWaiting(fd, readFlags)
182 return fd, 'r', nil
183 }
184
185 func (p *pollster) Close() error {
186     return os.NewSyscallError("close", syscall.Close(p.e
187 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/file.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package net
8
9 import (
10     "os"
11     "syscall"
12 )
13
14 func newFileFD(f *os.File) (*netFD, error) {
15     fd, err := syscall.Dup(int(f.Fd()))
16     if err != nil {
17         return nil, os.NewSyscallError("dup", err)
18     }
19
20     proto, err := syscall.GetsockoptInt(fd, syscall.SOL_
21 if err != nil {
22         return nil, os.NewSyscallError("getsockopt",
23 }
24
25     family := syscall.AF_UNSPEC
26     toAddr := sockaddrToTCP
27     sa, _ := syscall.Getsockname(fd)
28     switch sa.(type) {
29     default:
30         closesocket(fd)
31         return nil, syscall.EINVAL
32     case *syscall.SockaddrInet4:
33         family = syscall.AF_INET
34         if proto == syscall.SOCK_DGRAM {
35             toAddr = sockaddrToUDP
36         } else if proto == syscall.SOCK_RAW {
37             toAddr = sockaddrToIP
38         }
39     case *syscall.SockaddrInet6:
40         family = syscall.AF_INET6
41         if proto == syscall.SOCK_DGRAM {
42             toAddr = sockaddrToUDP
43         } else if proto == syscall.SOCK_RAW {
44             toAddr = sockaddrToIP
```

```

45     }
46     case *syscall.SockaddrUnix:
47         family = syscall.AF_UNIX
48         toAddr = sockaddrToUnix
49         if proto == syscall.SOCK_DGRAM {
50             toAddr = sockaddrToUnixgram
51         } else if proto == syscall.SOCK_SEQPACKET {
52             toAddr = sockaddrToUnixpacket
53         }
54     }
55     laddr := toAddr(sa)
56     sa, _ = syscall.Getpeername(fd)
57     raddr := toAddr(sa)
58
59     netfd, err := newFD(fd, family, proto, laddr.Network)
60     if err != nil {
61         return nil, err
62     }
63     netfd.setAddr(laddr, raddr)
64     return netfd, nil
65 }
66
67 // FileConn returns a copy of the network connection corresp
68 // the open file f. It is the caller's responsibility to cl
69 // finished. Closing c does not affect f, and closing f doe
70 // affect c.
71 func FileConn(f *os.File) (c Conn, err error) {
72     fd, err := newFileFD(f)
73     if err != nil {
74         return nil, err
75     }
76     switch fd.laddr.(type) {
77     case *TCPAddr:
78         return newTCPConn(fd), nil
79     case *UDPAddr:
80         return newUDPCConn(fd), nil
81     case *UnixAddr:
82         return newUnixConn(fd), nil
83     case *IPAddr:
84         return newIPConn(fd), nil
85     }
86     fd.Close()
87     return nil, syscall.EINVAL
88 }
89
90 // FileListener returns a copy of the network listener corre
91 // to the open file f. It is the caller's responsibility to
92 // when finished. Closing c does not affect l, and closing
93 // affect c.
94 func FileListener(f *os.File) (l Listener, err error) {

```

```

95         fd, err := newFileFD(f)
96         if err != nil {
97             return nil, err
98         }
99         switch laddr := fd.laddr.(type) {
100        case *TCPAddr:
101            return &TCPListener{fd}, nil
102        case *UnixAddr:
103            return &UnixListener{fd, laddr.Name}, nil
104        }
105        fd.Close()
106        return nil, syscall.EINVAL
107    }
108
109    // FilePacketConn returns a copy of the packet network connection
110    // corresponding to the open file f. It is the caller's
111    // responsibility to close f when finished. Closing c does
112    // not close f, and closing f does not affect c.
113    func FilePacketConn(f *os.File) (c PacketConn, err error) {
114        fd, err := newFileFD(f)
115        if err != nil {
116            return nil, err
117        }
118        switch fd.laddr.(type) {
119        case *UDPAddr:
120            return newUDPCConn(fd), nil
121        case *UnixAddr:
122            return newUnixConn(fd), nil
123        }
124        fd.Close()
125        return nil, syscall.EINVAL
126    }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/hosts.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Read static host/IP entries from /etc/hosts.
6
7 package net
8
9 import (
10     "sync"
11     "time"
12 )
13
14 const cacheMaxAge = 5 * time.Minute
15
16 // hostsPath points to the file with static IP/address entri
17 var hostsPath = "/etc/hosts"
18
19 // Simple cache.
20 var hosts struct {
21     sync.Mutex
22     byName map[string][]string
23     byAddr map[string][]string
24     expire time.Time
25     path   string
26 }
27
28 func readHosts() {
29     now := time.Now()
30     hp := hostsPath
31     if len(hosts.byName) == 0 || now.After(hosts.expire)
32         hs := make(map[string][]string)
33         is := make(map[string][]string)
34         var file *file
35         if file, _ = open(hp); file == nil {
36             return
37         }
38         for line, ok := file.readLine(); ok; line, c
39             if i := byteIndex(line, '#'); i >= 0
40                 // Discard comments.
41                 line = line[0:i]
42             }
43             f := getFields(line)
44             if len(f) < 2 || ParseIP(f[0]) == ni
```

```

45             continue
46         }
47         for i := 1; i < len(f); i++ {
48             h := f[i]
49             hs[h] = append(hs[h], f[0])
50             is[f[0]] = append(is[f[0]],
51                 )
52         }
53         // Update the data cache.
54         hosts.expire = time.Now().Add(cacheMaxAge)
55         hosts.path = hp
56         hosts.byName = hs
57         hosts.byAddr = is
58         file.close()
59     }
60 }
61
62 // lookupStaticHost looks up the addresses for the given host
63 func lookupStaticHost(host string) []string {
64     hosts.Lock()
65     defer hosts.Unlock()
66     readHosts()
67     if len(hosts.byName) != 0 {
68         if ips, ok := hosts.byName[host]; ok {
69             return ips
70         }
71     }
72     return nil
73 }
74
75 // lookupStaticAddr looks up the hosts for the given address
76 func lookupStaticAddr(addr string) []string {
77     hosts.Lock()
78     defer hosts.Unlock()
79     readHosts()
80     if len(hosts.byAddr) != 0 {
81         if hosts, ok := hosts.byAddr[addr]; ok {
82             return hosts
83         }
84     }
85     return nil
86 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/interface.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Network interface identification
6
7 package net
8
9 import "errors"
10
11 var (
12     errInvalidInterface      = errors.New("net: inval
13     errInvalidInterfaceIndex = errors.New("net: inval
14     errInvalidInterfaceName  = errors.New("net: inval
15     errNoSuchInterface       = errors.New("net: no su
16     errNoSuchMulticastInterface = errors.New("net: no su
17 )
18
19 // Interface represents a mapping between network interface
20 // and index. It also represents network interface facility
21 // information.
22 type Interface struct {
23     Index      int // positive integer that s
24     MTU        int // maximum transmission un
25     Name       string // e.g., "en0", "lo0", "et
26     HardwareAddr HardwareAddr // IEEE MAC-48, EUI-48 and
27     Flags      Flags // e.g., FlagUp, FlagLoopb
28 }
29
30 type Flags uint
31
32 const (
33     FlagUp          Flags = 1 << iota // interface is u
34     FlagBroadcast // interface supp
35     FlagLoopback // interface is a
36     FlagPointToPoint // interface belo
37     FlagMulticast // interface supp
38 )
39
40 var flagNames = []string{
41     "up",
42     "broadcast",
43     "loopback",
44     "pointtopoint",
```

```

45         "multicast",
46     }
47
48     func (f Flags) String() string {
49         s := ""
50         for i, name := range flagNames {
51             if f&(1<<uint(i)) != 0 {
52                 if s != "" {
53                     s += "|"
54                 }
55                 s += name
56             }
57         }
58         if s == "" {
59             s = "0"
60         }
61         return s
62     }
63
64     // Addr returns interface addresses for a specific interface
65     func (ifi *Interface) Addr() ([]Addr, error) {
66         if ifi == nil {
67             return nil, errInvalidInterface
68         }
69         return interfaceAddrTable(ifi.Index)
70     }
71
72     // MulticastAddr returns multicast, joined group addresses
73     // a specific interface.
74     func (ifi *Interface) MulticastAddr() ([]Addr, error) {
75         if ifi == nil {
76             return nil, errInvalidInterface
77         }
78         return interfaceMulticastAddrTable(ifi.Index)
79     }
80
81     // Interfaces returns a list of the system's network interfaces
82     func Interfaces() ([]Interface, error) {
83         return interfaceTable(0)
84     }
85
86     // InterfaceAddr returns a list of the system's network interface
87     // addresses.
88     func InterfaceAddr() ([]Addr, error) {
89         return interfaceAddrTable(0)
90     }
91
92     // InterfaceByIndex returns the interface specified by index
93     func InterfaceByIndex(index int) (*Interface, error) {
94         if index <= 0 {

```

```

95         return nil, errInvalidInterfaceIndex
96     }
97     ift, err := interfaceTable(index)
98     if err != nil {
99         return nil, err
100    }
101    for _, ifi := range ift {
102        return &ifi, nil
103    }
104    return nil, errNoSuchInterface
105 }
106
107 // InterfaceByName returns the interface specified by name.
108 func InterfaceByName(name string) (*Interface, error) {
109     if name == "" {
110         return nil, errInvalidInterfaceName
111     }
112     ift, err := interfaceTable(0)
113     if err != nil {
114         return nil, err
115     }
116     for _, ifi := range ift {
117         if name == ifi.Name {
118             return &ifi, nil
119         }
120     }
121     return nil, errNoSuchInterface
122 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/interface_linux.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Network interface identification for Linux
6
7 package net
8
9 import (
10     "os"
11     "syscall"
12     "unsafe"
13 )
14
15 // If the ifindex is zero, interfaceTable returns mappings of
16 // network interfaces. Otherwise it returns a mapping of a
17 // interface.
18 func interfaceTable(ifindex int) ([]Interface, error) {
19     tab, err := syscall.NetlinkRIB(syscall.RTM_GETLINK,
20     if err != nil {
21         return nil, os.NewSyscallError("netlink rib"
22     }
23
24     msgs, err := syscall.ParseNetlinkMessage(tab)
25     if err != nil {
26         return nil, os.NewSyscallError("netlink mess
27     }
28
29     var ift []Interface
30     for _, m := range msgs {
31         switch m.Header.Type {
32             case syscall.NLMSG_DONE:
33                 goto done
34             case syscall.RTM_NEWLINK:
35                 ifim := (*syscall.IfInfomsg)(unsafe.
36                 if ifindex == 0 || ifindex == int(if
37                     attrs, err := syscall.ParseN
38                     if err != nil {
39                         return nil, os.NewSy
40                     }
41                 ifi := newLink(ifim, attrs)
```

```

42             ift = append(ift, ifi)
43         }
44     }
45 }
46 done:
47     return ift, nil
48 }
49
50 func newLink(ifim *syscall.IfInfomsg, attrs []syscall.Netlin
51     ifi := Interface{Index: int(ifim.Index), Flags: link
52     for _, a := range attrs {
53         switch a.Attr.Type {
54             case syscall.IFLA_ADDRESS:
55                 var nonzero bool
56                 for _, b := range a.Value {
57                     if b != 0 {
58                         nonzero = true
59                     }
60                 }
61                 if nonzero {
62                     ifi.HardwareAddr = a.Value[:
63                 }
64             case syscall.IFLA_IFNAME:
65                 ifi.Name = string(a.Value[:len(a.Val
66             case syscall.IFLA_MTU:
67                 ifi.MTU = int(uint32(a.Value[3])<<24
68         }
69     }
70     return ifi
71 }
72
73 func linkFlags(rawFlags uint32) Flags {
74     var f Flags
75     if rawFlags&syscall.IFF_UP != 0 {
76         f |= FlagUp
77     }
78     if rawFlags&syscall.IFF_BROADCAST != 0 {
79         f |= FlagBroadcast
80     }
81     if rawFlags&syscall.IFF_LOOPBACK != 0 {
82         f |= FlagLoopback
83     }
84     if rawFlags&syscall.IFF_POINTOPOINT != 0 {
85         f |= FlagPointToPoint
86     }
87     if rawFlags&syscall.IFF_MULTICAST != 0 {
88         f |= FlagMulticast
89     }
90     return f
91 }

```

```

92
93 // If the ifindex is zero, interfaceAddrTable returns address
94 // for all network interfaces. Otherwise it returns address
95 // for a specific interface.
96 func interfaceAddrTable(ifindex int) ([]Addr, error) {
97     tab, err := syscall.NetlinkRIB(syscall.RTM_GETADDR,
98     if err != nil {
99         return nil, os.NewSyscallError("netlink rib"
100     }
101
102     msgs, err := syscall.ParseNetlinkMessage(tab)
103     if err != nil {
104         return nil, os.NewSyscallError("netlink mess
105     }
106
107     ifat, err := addrTable(msgs, ifindex)
108     if err != nil {
109         return nil, err
110     }
111     return ifat, nil
112 }
113
114 func addrTable(msgs []syscall.NetlinkMessage, ifindex int) (
115     var ifat []Addr
116     for _, m := range msgs {
117         switch m.Header.Type {
118             case syscall.NLMSG_DONE:
119                 goto done
120             case syscall.RTM_NEWADDR:
121                 ifam := (*syscall.IfAddrmsg)(unsafe.
122                 if ifindex == 0 || ifindex == int(if
123                 attrs, err := syscall.ParseN
124                 if err != nil {
125                     return nil, os.NewSy
126                 }
127                 ifat = append(ifat, newAddr(
128             }
129         }
130     }
131 done:
132     return ifat, nil
133 }
134
135 func newAddr(attrs []syscall.NetlinkRouteAttr, family, pfxle
136     ifa := &IPNet{}
137     for _, a := range attrs {
138         switch a.Attr.Type {
139             case syscall.IFA_ADDRESS:
140                 switch family {

```

```

141         case syscall.AF_INET:
142             ifa.IP = IPv4(a.Value[0], a.
143                 ifa.Mask = CIDRMask(pfxlen,
144         case syscall.AF_INET6:
145             ifa.IP = make(IP, IPv6len)
146             copy(ifa.IP, a.Value[:])
147             ifa.Mask = CIDRMask(pfxlen,
148         }
149     }
150 }
151     return ifa
152 }
153
154 // If the ifindex is zero, interfaceMulticastAddrTable retur
155 // addresses for all network interfaces. Otherwise it retur
156 // addresses for a specific interface.
157 func interfaceMulticastAddrTable(ifindex int) ([]Addr, error
158     var (
159         err error
160         ifi *Interface
161     )
162     if ifindex > 0 {
163         ifi, err = InterfaceByIndex(ifindex)
164         if err != nil {
165             return nil, err
166         }
167     }
168     ifmat4 := parseProcNetIGMP("/proc/net/igmp", ifi)
169     ifmat6 := parseProcNetIGMP6("/proc/net/igmp6", ifi)
170     return append(ifmat4, ifmat6...), nil
171 }
172
173 func parseProcNetIGMP(path string, ifi *Interface) []Addr {
174     fd, err := open(path)
175     if err != nil {
176         return nil
177     }
178     defer fd.close()
179
180     var (
181         ifmat []Addr
182         name string
183     )
184     fd.readLine() // skip first line
185     b := make([]byte, IPv4len)
186     for l, ok := fd.readLine(); ok; l, ok = fd.readLine(
187         f := splitAtBytes(l, " :\r\t\n")
188         if len(f) < 4 {
189             continue

```

```

190     }
191     switch {
192     case l[0] != ' ' && l[0] != '\t': // new int
193         name = f[1]
194     case len(f[0]) == 8:
195         if ifi == nil || name == ifi.Name {
196             for i := 0; i+1 < len(f[0]);
197                 b[i/2], _ = xtoi2(f[
198                 }
199                 ifma := IPAddr{IP: IPv4(b[3]
200                 ifmat = append(ifmat, ifma.t
201             }
202         }
203     }
204     return ifmat
205 }
206
207 func parseProcNetIGMP6(path string, ifi *Interface) []Addr {
208     fd, err := open(path)
209     if err != nil {
210         return nil
211     }
212     defer fd.close()
213
214     var ifmat []Addr
215     b := make([]byte, IPv6len)
216     for l, ok := fd.readLine(); ok; l, ok = fd.readLine(
217         f := splitAtBytes(l, "\r\t\n")
218         if len(f) < 6 {
219             continue
220         }
221         if ifi == nil || f[1] == ifi.Name {
222             for i := 0; i+1 < len(f[2]); i += 2
223                 b[i/2], _ = xtoi2(f[2][i:i+2
224             }
225             ifma := IPAddr{IP: IP{b[0], b[1], b[
226             ifmat = append(ifmat, ifma.toAddr())
227         }
228     }
229     return ifmat
230 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/ip.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // IP address manipulations
6 //
7 // IPv4 addresses are 4 bytes; IPv6 addresses are 16 bytes.
8 // An IPv4 address can be converted to an IPv6 address by
9 // adding a canonical prefix (10 zeros, 2 0xFFs).
10 // This library accepts either size of byte array but always
11 // returns 16-byte addresses.
12
13 package net
14
15 // IP address lengths (bytes).
16 const (
17     IPv4len = 4
18     IPv6len = 16
19 )
20
21 // An IP is a single IP address, an array of bytes.
22 // Functions in this package accept either 4-byte (IPv4)
23 // or 16-byte (IPv6) arrays as input.
24 //
25 // Note that in this documentation, referring to an
26 // IP address as an IPv4 address or an IPv6 address
27 // is a semantic property of the address, not just the
28 // length of the byte array: a 16-byte array can still
29 // be an IPv4 address.
30 type IP []byte
31
32 // An IP mask is an IP address.
33 type IPMask []byte
34
35 // An IPNet represents an IP network.
36 type IPNet struct {
37     IP      IP      // network number
38     Mask    IPMask // network mask
39 }
40
41 // IPv4 returns the IP address (in 16-byte form) of the
42 // IPv4 address a.b.c.d.
43 func IPv4(a, b, c, d byte) IP {
44     p := make(IP, IPv6len)
```

```

45         copy(p, v4InV6Prefix)
46         p[12] = a
47         p[13] = b
48         p[14] = c
49         p[15] = d
50         return p
51     }
52
53     var v4InV6Prefix = []byte{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0xff
54
55     // IPv4Mask returns the IP mask (in 4-byte form) of the
56     // IPv4 mask a.b.c.d.
57     func IPv4Mask(a, b, c, d byte) IPMask {
58         p := make(IPMask, IPv4len)
59         p[0] = a
60         p[1] = b
61         p[2] = c
62         p[3] = d
63         return p
64     }
65
66     // CIDRMask returns an IPMask consisting of `ones' 1 bits
67     // followed by 0s up to a total length of `bits' bits.
68     // For a mask of this form, CIDRMask is the inverse of IPMas
69     func CIDRMask(ones, bits int) IPMask {
70         if bits != 8*IPv4len && bits != 8*IPv6len {
71             return nil
72         }
73         if ones < 0 || ones > bits {
74             return nil
75         }
76         l := bits / 8
77         m := make(IPMask, l)
78         n := uint(ones)
79         for i := 0; i < l; i++ {
80             if n >= 8 {
81                 m[i] = 0xff
82                 n -= 8
83                 continue
84             }
85             m[i] = ^byte(0xff >> n)
86             n = 0
87         }
88         return m
89     }
90
91     // Well-known IPv4 addresses
92     var (
93         IPv4bcast      = IPv4(255, 255, 255, 255) // broadcas
94         IPv4allsys     = IPv4(224, 0, 0, 1)       // all syst

```

```

95         IPv4allrouter = IPv4(224, 0, 0, 2)           // all rout
96         IPv4zero      = IPv4(0, 0, 0, 0)           // all zero
97     )
98
99     // Well-known IPv6 addresses
100    var (
101        IPv6zero          = IP{0, 0, 0, 0, 0, 0, 0,
102        IPv6unspecified   = IP{0, 0, 0, 0, 0, 0, 0,
103        IPv6loopback      = IP{0, 0, 0, 0, 0, 0, 0,
104        IPv6interfacelocalallnodes = IP{0xff, 0x01, 0, 0, 0,
105        IPv6linklocalallnodes = IP{0xff, 0x02, 0, 0, 0,
106        IPv6linklocalallrouters = IP{0xff, 0x02, 0, 0, 0,
107    )
108
109    // IsUnspecified returns true if ip is an unspecified address
110    func (ip IP) IsUnspecified() bool {
111        if ip.Equal(IPv4zero) || ip.Equal(IPv6unspecified) {
112            return true
113        }
114        return false
115    }
116
117    // IsLoopback returns true if ip is a loopback address.
118    func (ip IP) IsLoopback() bool {
119        if ip4 := ip.To4(); ip4 != nil && ip4[0] == 127 {
120            return true
121        }
122        return ip.Equal(IPv6loopback)
123    }
124
125    // IsMulticast returns true if ip is a multicast address.
126    func (ip IP) IsMulticast() bool {
127        if ip4 := ip.To4(); ip4 != nil && ip4[0]&0xf0 == 0xe
128            return true
129        }
130        return ip[0] == 0xff
131    }
132
133    // IsInterfaceLinkLocalMulticast returns true if ip is
134    // an interface-local multicast address.
135    func (ip IP) IsInterfaceLocalMulticast() bool {
136        return len(ip) == IPv6len && ip[0] == 0xff && ip[1]&
137    }
138
139    // IsLinkLocalMulticast returns true if ip is a link-local
140    // multicast address.
141    func (ip IP) IsLinkLocalMulticast() bool {
142        if ip4 := ip.To4(); ip4 != nil && ip4[0] == 224 && i
143            return true

```

```

144     }
145     return ip[0] == 0xff && ip[1]&0x0f == 0x02
146 }
147
148 // IsLinkLocalUnicast returns true if ip is a link-local
149 // unicast address.
150 func (ip IP) IsLinkLocalUnicast() bool {
151     if ip4 := ip.To4(); ip4 != nil && ip4[0] == 169 && i
152         return true
153     }
154     return ip[0] == 0xfe && ip[1]&0xc0 == 0x80
155 }
156
157 // IsGlobalUnicast returns true if ip is a global unicast
158 // address.
159 func (ip IP) IsGlobalUnicast() bool {
160     return !ip.IsUnspecified() &&
161         !ip.IsLoopback() &&
162         !ip.IsMulticast() &&
163         !ip.IsLinkLocalUnicast()
164 }
165
166 // Is p all zeros?
167 func isZeros(p IP) bool {
168     for i := 0; i < len(p); i++ {
169         if p[i] != 0 {
170             return false
171         }
172     }
173     return true
174 }
175
176 // To4 converts the IPv4 address ip to a 4-byte representati
177 // If ip is not an IPv4 address, To4 returns nil.
178 func (ip IP) To4() IP {
179     if len(ip) == IPv4len {
180         return ip
181     }
182     if len(ip) == IPv6len &&
183         isZeros(ip[0:10]) &&
184         ip[10] == 0xff &&
185         ip[11] == 0xff {
186         return ip[12:16]
187     }
188     return nil
189 }
190
191 // To16 converts the IP address ip to a 16-byte representati
192 // If ip is not an IP address (it is the wrong length), To16

```

```

193 func (ip IP) To16() IP {
194     if len(ip) == IPv4len {
195         return IPv4(ip[0], ip[1], ip[2], ip[3])
196     }
197     if len(ip) == IPv6len {
198         return ip
199     }
200     return nil
201 }
202
203 // Default route masks for IPv4.
204 var (
205     classAMask = IPv4Mask(0xff, 0, 0, 0)
206     classBMask = IPv4Mask(0xff, 0xff, 0, 0)
207     classCMask = IPv4Mask(0xff, 0xff, 0xff, 0)
208 )
209
210 // DefaultMask returns the default IP mask for the IP address
211 // Only IPv4 addresses have default masks; DefaultMask returns
212 // nil if ip is not a valid IPv4 address.
213 func (ip IP) DefaultMask() IPMask {
214     if ip = ip.To4(); ip == nil {
215         return nil
216     }
217     switch true {
218     case ip[0] < 0x80:
219         return classAMask
220     case ip[0] < 0xC0:
221         return classBMask
222     default:
223         return classCMask
224     }
225     return nil // not reached
226 }
227
228 func allFF(b []byte) bool {
229     for _, c := range b {
230         if c != 0xff {
231             return false
232         }
233     }
234     return true
235 }
236
237 // Mask returns the result of masking the IP address ip with
238 func (ip IP) Mask(mask IPMask) IP {
239     if len(mask) == IPv6len && len(ip) == IPv4len && all
240         mask = mask[12:]
241     }
242     if len(mask) == IPv4len && len(ip) == IPv6len && byt

```

```

243         ip = ip[12:]
244     }
245     n := len(ip)
246     if n != len(mask) {
247         return nil
248     }
249     out := make(IP, n)
250     for i := 0; i < n; i++ {
251         out[i] = ip[i] & mask[i]
252     }
253     return out
254 }
255
256 // String returns the string form of the IP address ip.
257 // If the address is an IPv4 address, the string representat
258 // is dotted decimal ("74.125.19.99"). Otherwise the repres
259 // is IPv6 ("2001:4860:0:2001::68").
260 func (ip IP) String() string {
261     p := ip
262
263     if len(ip) == 0 {
264         return "<nil>"
265     }
266
267     // If IPv4, use dotted notation.
268     if p4 := p.To4(); len(p4) == IPv4len {
269         return itod(uint(p4[0])) + "." +
270             itod(uint(p4[1])) + "." +
271             itod(uint(p4[2])) + "." +
272             itod(uint(p4[3]))
273     }
274     if len(p) != IPv6len {
275         return "?"
276     }
277
278     // Find longest run of zeros.
279     e0 := -1
280     e1 := -1
281     for i := 0; i < IPv6len; i += 2 {
282         j := i
283         for j < IPv6len && p[j] == 0 && p[j+1] == 0
284             j += 2
285     }
286     if j > i && j-i > e1-e0 {
287         e0 = i
288         e1 = j
289     }
290 }
291 // The symbol "::" MUST NOT be used to shorten just

```

```

292         if e1-e0 <= 2 {
293             e0 = -1
294             e1 = -1
295         }
296
297         // Print with possible :: in place of run of zeros
298         var s string
299         for i := 0; i < IPv6len; i += 2 {
300             if i == e0 {
301                 s += "::"
302                 i = e1
303                 if i >= IPv6len {
304                     break
305                 }
306             } else if i > 0 {
307                 s += ":"
308             }
309             s += itox((uint(p[i])<<8)|uint(p[i+1]), 1)
310         }
311         return s
312     }
313
314     // Equal returns true if ip and x are the same IP address.
315     // An IPv4 address and that same address in IPv6 form are
316     // considered to be equal.
317     func (ip IP) Equal(x IP) bool {
318         if len(ip) == len(x) {
319             return bytesEqual(ip, x)
320         }
321         if len(ip) == IPv4len && len(x) == IPv6len {
322             return bytesEqual(x[0:12], v4InV6Prefix) &&
323         }
324         if len(ip) == IPv6len && len(x) == IPv4len {
325             return bytesEqual(ip[0:12], v4InV6Prefix) &&
326         }
327         return false
328     }
329
330     func bytesEqual(x, y []byte) bool {
331         if len(x) != len(y) {
332             return false
333         }
334         for i, b := range x {
335             if y[i] != b {
336                 return false
337             }
338         }
339         return true
340     }

```

```

341
342 // If mask is a sequence of 1 bits followed by 0 bits,
343 // return the number of 1 bits.
344 func simpleMaskLength(mask IPMask) int {
345     var n int
346     for i, v := range mask {
347         if v == 0xff {
348             n += 8
349             continue
350         }
351         // found non-ff byte
352         // count 1 bits
353         for v&0x80 != 0 {
354             n++
355             v <<= 1
356         }
357         // rest must be 0 bits
358         if v != 0 {
359             return -1
360         }
361         for i++; i < len(mask); i++ {
362             if mask[i] != 0 {
363                 return -1
364             }
365         }
366         break
367     }
368     return n
369 }
370
371 // Size returns the number of leading ones and total bits in
372 // If the mask is not in the canonical form--ones followed b
373 // Size returns 0, 0.
374 func (m IPMask) Size() (ones, bits int) {
375     ones, bits = simpleMaskLength(m), len(m)*8
376     if ones == -1 {
377         return 0, 0
378     }
379     return
380 }
381
382 // String returns the hexadecimal form of m, with no punctua
383 func (m IPMask) String() string {
384     s := ""
385     for _, b := range m {
386         s += itox(uint(b), 2)
387     }
388     if len(s) == 0 {
389         return "<nil>"
390     }

```

```

391         return s
392     }
393
394     func networkNumberAndMask(n *IPNet) (ip IP, m IPMask) {
395         if ip = n.IP.To4(); ip == nil {
396             ip = n.IP
397             if len(ip) != IPv6len {
398                 return nil, nil
399             }
400         }
401         m = n.Mask
402         switch len(m) {
403         case IPv4len:
404             if len(ip) != IPv4len {
405                 return nil, nil
406             }
407         case IPv6len:
408             if len(ip) == IPv4len {
409                 m = m[12:]
410             }
411         default:
412             return nil, nil
413         }
414         return
415     }
416
417     // Contains reports whether the network includes ip.
418     func (n *IPNet) Contains(ip IP) bool {
419         nn, m := networkNumberAndMask(n)
420         if x := ip.To4(); x != nil {
421             ip = x
422         }
423         l := len(ip)
424         if l != len(nn) {
425             return false
426         }
427         for i := 0; i < l; i++ {
428             if nn[i]&m[i] != ip[i]&m[i] {
429                 return false
430             }
431         }
432         return true
433     }
434
435     // String returns the CIDR notation of n like "192.168.100.1
436     // or "2001:DB8::/48" as defined in RFC 4632 and RFC 4291.
437     // If the mask is not in the canonical form, it returns the
438     // string which consists of an IP address, followed by a sla
439     // character and a mask expressed as hexadecimal form with n

```

```

440 // punctuation like "192.168.100.1/c000ff00".
441 func (n *IPNet) String() string {
442     nn, m := networkNumberAndMask(n)
443     if nn == nil || m == nil {
444         return "<nil>"
445     }
446     l := simpleMaskLength(m)
447     if l == -1 {
448         return nn.String() + "/" + m.String()
449     }
450     return nn.String() + "/" + itod(uint(l))
451 }
452
453 // Network returns the address's network name, "ip+net".
454 func (n *IPNet) Network() string { return "ip+net" }
455
456 // Parse IPv4 address (d.d.d.d).
457 func parseIPv4(s string) IP {
458     var p [IPv4len]byte
459     i := 0
460     for j := 0; j < IPv4len; j++ {
461         if i >= len(s) {
462             // Missing octets.
463             return nil
464         }
465         if j > 0 {
466             if s[i] != '.' {
467                 return nil
468             }
469             i++
470         }
471         var (
472             n int
473             ok bool
474         )
475         n, i, ok = dtoi(s, i)
476         if !ok || n > 0xFF {
477             return nil
478         }
479         p[j] = byte(n)
480     }
481     if i != len(s) {
482         return nil
483     }
484     return IPv4(p[0], p[1], p[2], p[3])
485 }
486
487 // Parse IPv6 address. Many forms.
488 // The basic form is a sequence of eight colon-separated

```

```

489 // 16-bit hex numbers separated by colons,
490 // as in 0123:4567:89ab:cdef:0123:4567:89ab:cdef.
491 // Two exceptions:
492 //     * A run of zeros can be replaced with "::".
493 //     * The last 32 bits can be in IPv4 form.
494 // Thus, ::ffff:1.2.3.4 is the IPv4 address 1.2.3.4.
495 func parseIPv6(s string) IP {
496     p := make(IP, IPv6len)
497     ellipsis := -1 // position of ellipsis in p
498     i := 0        // index in string s
499
500     // Might have leading ellipsis
501     if len(s) >= 2 && s[0] == ':' && s[1] == ':' {
502         ellipsis = 0
503         i = 2
504         // Might be only ellipsis
505         if i == len(s) {
506             return p
507         }
508     }
509
510     // Loop, parsing hex numbers followed by colon.
511     j := 0
512     for j < IPv6len {
513         // Hex number.
514         n, i1, ok := xtoi(s, i)
515         if !ok || n > 0xFFFF {
516             return nil
517         }
518
519         // If followed by dot, might be in trailing
520         if i1 < len(s) && s[i1] == '.' {
521             if ellipsis < 0 && j != IPv6len-IPv4 {
522                 // Not the right place.
523                 return nil
524             }
525             if j+IPv4len > IPv6len {
526                 // Not enough room.
527                 return nil
528             }
529             p4 := parseIPv4(s[i:])
530             if p4 == nil {
531                 return nil
532             }
533             p[j] = p4[12]
534             p[j+1] = p4[13]
535             p[j+2] = p4[14]
536             p[j+3] = p4[15]
537             i = len(s)
538             j += IPv4len

```

```

539             break
540         }
541
542         // Save this 16-bit chunk.
543         p[j] = byte(n >> 8)
544         p[j+1] = byte(n)
545         j += 2
546
547         // Stop at end of string.
548         i = i1
549         if i == len(s) {
550             break
551         }
552
553         // Otherwise must be followed by colon and n
554         if s[i] != ':' || i+1 == len(s) {
555             return nil
556         }
557         i++
558
559         // Look for ellipsis.
560         if s[i] == ':' {
561             if ellipsis >= 0 { // already have o
562                 return nil
563             }
564             ellipsis = j
565             if i++; i == len(s) { // can be at e
566                 break
567             }
568         }
569     }
570
571     // Must have used entire string.
572     if i != len(s) {
573         return nil
574     }
575
576     // If didn't parse enough, expand ellipsis.
577     if j < IPv6len {
578         if ellipsis < 0 {
579             return nil
580         }
581         n := IPv6len - j
582         for k := j - 1; k >= ellipsis; k-- {
583             p[k+n] = p[k]
584         }
585         for k := ellipsis + n - 1; k >= ellipsis; k-
586             p[k] = 0
587     }

```

```

588         }
589         return p
590     }
591
592     // A ParseError represents a malformed text string and the t
593     type ParseError struct {
594         Type string
595         Text string
596     }
597
598     func (e *ParseError) Error() string {
599         return "invalid " + e.Type + ": " + e.Text
600     }
601
602     func parseIP(s string) IP {
603         if p := parseIPv4(s); p != nil {
604             return p
605         }
606         if p := parseIPv6(s); p != nil {
607             return p
608         }
609         return nil
610     }
611
612     // ParseIP parses s as an IP address, returning the result.
613     // The string s can be in dotted decimal ("74.125.19.99")
614     // or IPv6 ("2001:4860:0:2001::68") form.
615     // If s is not a valid textual representation of an IP address,
616     // ParseIP returns nil.
617     func ParseIP(s string) IP {
618         if p := parseIPv4(s); p != nil {
619             return p
620         }
621         return parseIPv6(s)
622     }
623
624     // ParseCIDR parses s as a CIDR notation IP address and mask
625     // like "192.168.100.1/24" or "2001:DB8::/48", as defined in
626     // RFC 4632 and RFC 4291.
627     //
628     // It returns the IP address and the network implied by the
629     // and mask. For example, ParseCIDR("192.168.100.1/16") returns
630     // the IP address 192.168.100.1 and the network 192.168.0.0/16.
631     func ParseCIDR(s string) (IP, *IPNet, error) {
632         i := byteIndex(s, '/')
633         if i < 0 {
634             return nil, nil, &ParseError{"CIDR address",
635
636                 ipstr, maskstr := s[:i], s[i+1:]

```

```
637         iplen := IPv4len
638         ip := parseIPv4(ipstr)
639         if ip == nil {
640             iplen = IPv6len
641             ip = parseIPv6(ipstr)
642         }
643         n, i, ok := dtoi(maskstr, 0)
644         if ip == nil || !ok || i != len(maskstr) || n < 0 ||
645             return nil, nil, &ParseError{"CIDR address",
646         }
647         m := CIDRMask(n, 8*iplen)
648         return ip, &IPNet{ip.Mask(m), m}, nil
649     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/iprawsock.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // (Raw) IP sockets
6
7 package net
8
9 // IPAddr represents the address of a IP end point.
10 type IPAddr struct {
11     IP IP
12 }
13
14 // Network returns the address's network name, "ip".
15 func (a *IPAddr) Network() string { return "ip" }
16
17 func (a *IPAddr) String() string {
18     if a == nil {
19         return "<nil>"
20     }
21     return a.IP.String()
22 }
23
24 // ResolveIPAddr parses addr as a IP address and resolves do
25 // names to numeric addresses on the network net, which must
26 // "ip", "ip4" or "ip6". A literal IPv6 host address must b
27 // enclosed in square brackets, as in "[::]".
28 func ResolveIPAddr(net, addr string) (*IPAddr, error) {
29     ip, err := hostToIP(net, addr)
30     if err != nil {
31         return nil, err
32     }
33     return &IPAddr{ip}, nil
34 }
35
36 // Convert "host" into IP address.
37 func hostToIP(net, host string) (ip IP, err error) {
38     var addr IP
39     // Try as an IP address.
40     addr = ParseIP(host)
41     if addr == nil {
42         filter := anyaddr
43         if net != "" && net[len(net)-1] == '4' {
44             filter = ipv4only
```

```

45     }
46     if net != "" && net[len(net)-1] == '6' {
47         filter = ipv6only
48     }
49     // Not an IP address. Try as a DNS name.
50     addrs, err1 := LookupHost(host)
51     if err1 != nil {
52         err = err1
53         goto Error
54     }
55     addr = firstFavoriteAddr(filter, addrs)
56     if addr == nil {
57         // should not happen
58         err = &AddrError{"LookupHost returned"}
59         goto Error
60     }
61 }
62 return addr, nil
63 Error:
64 return nil, err
65 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/iprawsock_posix.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 // (Raw) IP sockets
8
9 package net
10
11 import (
12     "os"
13     "syscall"
14     "time"
15 )
16
17 func sockaddrToIP(sa syscall.Sockaddr) Addr {
18     switch sa := sa.(type) {
19     case *syscall.SockaddrInet4:
20         return &IPAddr{sa.Addr[0:]}
21     case *syscall.SockaddrInet6:
22         return &IPAddr{sa.Addr[0:]}
23     }
24     return nil
25 }
26
27 func (a *IPAddr) family() int {
28     if a == nil || len(a.IP) <= IPv4len {
29         return syscall.AF_INET
30     }
31     if a.IP.To4() != nil {
32         return syscall.AF_INET
33     }
34     return syscall.AF_INET6
35 }
36
37 func (a *IPAddr) isWildcard() bool {
38     if a == nil || a.IP == nil {
39         return true
40     }
41     return a.IP.IsUnspecified()
```

```

42 }
43
44 func (a *IPAddr) sockaddr(family int) (syscall.Sockaddr, error) {
45     return ipToSockaddr(family, a.IP, 0)
46 }
47
48 func (a *IPAddr) toAddr() sockaddr {
49     if a == nil { // nil *IPAddr
50         return nil // nil interface
51     }
52     return a
53 }
54
55 // IPConn is the implementation of the Conn and PacketConn
56 // interfaces for IP network connections.
57 type IPConn struct {
58     fd *netFD
59 }
60
61 func newIPConn(fd *netFD) *IPConn { return &IPConn{fd} }
62
63 func (c *IPConn) ok() bool { return c != nil && c.fd != nil }
64
65 // Implementation of the Conn interface - see Conn for docum
66
67 // Read implements the Conn Read method.
68 func (c *IPConn) Read(b []byte) (int, error) {
69     n, _, err := c.ReadFrom(b)
70     return n, err
71 }
72
73 // Write implements the Conn Write method.
74 func (c *IPConn) Write(b []byte) (int, error) {
75     if !c.ok() {
76         return 0, syscall.EINVAL
77     }
78     return c.fd.Write(b)
79 }
80
81 // Close closes the IP connection.
82 func (c *IPConn) Close() error {
83     if !c.ok() {
84         return syscall.EINVAL
85     }
86     return c.fd.Close()
87 }
88
89 // LocalAddr returns the local network address.
90 func (c *IPConn) LocalAddr() Addr {
91     if !c.ok() {

```

```

92             return nil
93         }
94         return c.fd.laddr
95     }
96
97     // RemoteAddr returns the remote network address, a *IPAddr.
98     func (c *IPConn) RemoteAddr() Addr {
99         if !c.ok() {
100             return nil
101         }
102         return c.fd.raddr
103     }
104
105     // SetDeadline implements the Conn SetDeadline method.
106     func (c *IPConn) SetDeadline(t time.Time) error {
107         if !c.ok() {
108             return syscall.EINVAL
109         }
110         return setDeadline(c.fd, t)
111     }
112
113     // SetReadDeadline implements the Conn SetReadDeadline metho
114     func (c *IPConn) SetReadDeadline(t time.Time) error {
115         if !c.ok() {
116             return syscall.EINVAL
117         }
118         return setReadDeadline(c.fd, t)
119     }
120
121     // SetWriteDeadline implements the Conn SetWriteDeadline met
122     func (c *IPConn) SetWriteDeadline(t time.Time) error {
123         if !c.ok() {
124             return syscall.EINVAL
125         }
126         return setWriteDeadline(c.fd, t)
127     }
128
129     // SetReadBuffer sets the size of the operating system's
130     // receive buffer associated with the connection.
131     func (c *IPConn) SetReadBuffer(bytes int) error {
132         if !c.ok() {
133             return syscall.EINVAL
134         }
135         return setReadBuffer(c.fd, bytes)
136     }
137
138     // SetWriteBuffer sets the size of the operating system's
139     // transmit buffer associated with the connection.
140     func (c *IPConn) SetWriteBuffer(bytes int) error {

```

```

141         if !c.ok() {
142             return syscall.EINVAL
143         }
144         return setWriteBuffer(c.fd, bytes)
145     }
146
147     // IP-specific methods.
148
149     // ReadFromIP reads a IP packet from c, copying the payload
150     // It returns the number of bytes copied into b and the retu
151     // that was on the packet.
152     //
153     // ReadFromIP can be made to time out and return an error wi
154     // Timeout() == true after a fixed time limit; see SetDeadli
155     // SetReadDeadline.
156     func (c *IPConn) ReadFromIP(b []byte) (int, *IPAddr, error)
157     {
158         if !c.ok() {
159             return 0, nil, syscall.EINVAL
160         }
161         // TODO(cw,rsc): consider using readv if we know the
162         // type to avoid the header trim/copy
163         var addr *IPAddr
164         n, sa, err := c.fd.ReadFrom(b)
165         switch sa := sa.(type) {
166         case *syscall.SockaddrInet4:
167             addr = &IPAddr{sa.Addr[0:]}
168             if len(b) >= IPv4len { // discard ipv4 heade
169                 hsize := (int(b[0]) & 0xf) * 4
170                 copy(b, b[hsize:])
171                 n -= hsize
172             }
173         case *syscall.SockaddrInet6:
174             addr = &IPAddr{sa.Addr[0:]}
175         }
176         return n, addr, err
177     }
178
179     // ReadFrom implements the PacketConn ReadFrom method.
180     func (c *IPConn) ReadFrom(b []byte) (int, Addr, error) {
181         if !c.ok() {
182             return 0, nil, syscall.EINVAL
183         }
184         n, uaddr, err := c.ReadFromIP(b)
185         return n, uaddr.toAddr(), err
186     }
187
188     // WriteToIP writes a IP packet to addr via c, copying the p
189     // WriteToIP can be made to time out and return

```

```

190 // an error with Timeout() == true after a fixed time limit;
191 // see SetDeadline and SetWriteDeadline.
192 // On packet-oriented connections, write timeouts are rare.
193 func (c *IPConn) WriteToIP(b []byte, addr *IPAddr) (int, err
194     if !c.ok() {
195         return 0, syscall.EINVAL
196     }
197     sa, err := addr.sockaddr(c.fd.family)
198     if err != nil {
199         return 0, &OpError{"write", c.fd.net, addr,
200     }
201     return c.fd.WriteTo(b, sa)
202 }
203
204 // WriteTo implements the PacketConn WriteTo method.
205 func (c *IPConn) WriteTo(b []byte, addr Addr) (int, error) {
206     if !c.ok() {
207         return 0, syscall.EINVAL
208     }
209     a, ok := addr.(*IPAddr)
210     if !ok {
211         return 0, &OpError{"write", c.fd.net, addr,
212     }
213     return c.WriteToIP(b, a)
214 }
215
216 // DialIP connects to the remote address raddr on the networ
217 // which must be "ip", "ip4", or "ip6" followed by a colon a
218 func DialIP(netProto string, laddr, raddr *IPAddr) (*IPConn,
219     net, proto, err := parseDialNetwork(netProto)
220     if err != nil {
221         return nil, err
222     }
223     switch net {
224     case "ip", "ip4", "ip6":
225     default:
226         return nil, UnknownNetworkError(net)
227     }
228     if raddr == nil {
229         return nil, &OpError{"dial", netProto, nil,
230     }
231     fd, err := internetSocket(net, laddr.toAddr(), raddr
232     if err != nil {
233         return nil, err
234     }
235     return newIPConn(fd), nil
236 }
237
238 // ListenIP listens for incoming IP packets addressed to the
239 // local address laddr. The returned connection c's ReadFro

```

```

240 // and WriteTo methods can be used to receive and send IP
241 // packets with per-packet addressing.
242 func ListenIP(netProto string, laddr *IPAddr) (*IPConn, error) {
243     net, proto, err := parseDialNetwork(netProto)
244     if err != nil {
245         return nil, err
246     }
247     switch net {
248     case "ip", "ip4", "ip6":
249     default:
250         return nil, UnknownNetworkError(net)
251     }
252     fd, err := internetSocket(net, laddr.toAddr(), nil,
253     if err != nil {
254         return nil, err
255     }
256     return newIPConn(fd), nil
257 }
258
259 // File returns a copy of the underlying os.File, set to blo
260 // It is the caller's responsibility to close f when finishe
261 // Closing c does not affect f, and closing f does not affec
262 func (c *IPConn) File() (f *os.File, error) { return c.f

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/ipsock.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // IP sockets
6
7 package net
8
9 var supportsIPv6, supportsIPv4map = probeIPv6Stack()
10
11 func firstFavoriteAddr(filter func(IP) IP, addrs []string) (
12     if filter == nil {
13         // We'll take any IP address, but since the
14         // does not yet try multiple addresses, pref
15         // an IPv4 address if possible. This is esp
16         // if localhost resolves to [ipv6-localhost,
17         // Too much code assumes localhost == ipv4-1
18         addr = firstSupportedAddr(ipv4only, addrs)
19         if addr == nil {
20             addr = firstSupportedAddr(anyaddr, a
21         }
22     } else {
23         addr = firstSupportedAddr(filter, addrs)
24     }
25     return
26 }
27
28 func firstSupportedAddr(filter func(IP) IP, addrs []string)
29     for _, s := range addrs {
30         if addr := filter(ParseIP(s)); addr != nil {
31             return addr
32         }
33     }
34     return nil
35 }
36
37 func anyaddr(x IP) IP {
38     if x4 := x.To4(); x4 != nil {
39         return x4
40     }
41     if supportsIPv6 {
42         return x
43     }
44     return nil
```

```

45 }
46
47 func ipv4only(x IP) IP { return x.To4() }
48
49 func ipv6only(x IP) IP {
50     // Only return addresses that we can use
51     // with the kernel's IPv6 addressing modes.
52     if len(x) == IPv6len && x.To4() == nil && supportsIP
53         return x
54     }
55     return nil
56 }
57
58 type InvalidAddrError string
59
60 func (e InvalidAddrError) Error() string { return string(e)
61 func (e InvalidAddrError) Timeout() bool { return false }
62 func (e InvalidAddrError) Temporary() bool { return false }
63
64 // SplitHostPort splits a network address of the form
65 // "host:port" or "[host]:port" into host and port.
66 // The latter form must be used when host contains a colon.
67 func SplitHostPort(hostport string) (host, port string, err
68     // The port starts after the last colon.
69     i := last(hostport, ':')
70     if i < 0 {
71         err = &AddrError{"missing port in address",
72         return
73     }
74
75     host, port = hostport[0:i], hostport[i+1:]
76
77     // Can put brackets around host ...
78     if len(host) > 0 && host[0] == '[' && host[len(host)
79         host = host[1 : len(host)-1]
80     } else {
81         // ... but if there are no brackets, no colo
82         if byteIndex(host, ':') >= 0 {
83             err = &AddrError{"too many colons in
84             return
85         }
86     }
87     return
88 }
89
90 // JoinHostPort combines host and port into a network address
91 // of the form "host:port" or, if host contains a colon, "[h
92 func JoinHostPort(host, port string) string {
93     // If host has colons, have to bracket it.
94     if byteIndex(host, ':') >= 0 {

```

```

95         return "[" + host + "]:" + port
96     }
97     return host + ":" + port
98 }
99
100 // Convert "host:port" into IP address and port.
101 func hostPortToIP(net, hostport string) (ip IP, iport int, e
102     host, port, err := SplitHostPort(hostport)
103     if err != nil {
104         return nil, 0, err
105     }
106
107     var addr IP
108     if host != "" {
109         // Try as an IP address.
110         addr = ParseIP(host)
111         if addr == nil {
112             var filter func(IP) IP
113             if net != "" && net[len(net)-1] == '
114                 filter = ipv4only
115             }
116             if net != "" && net[len(net)-1] == '
117                 filter = ipv6only
118             }
119             // Not an IP address. Try as a DNS
120             addrs, err := LookupHost(host)
121             if err != nil {
122                 return nil, 0, err
123             }
124             addr = firstFavoriteAddr(filter, add
125             if addr == nil {
126                 // should not happen
127                 return nil, 0, &AddrError{"L
128             }
129         }
130     }
131
132     p, i, ok := dtoi(port, 0)
133     if !ok || i != len(port) {
134         p, err = LookupPort(net, port)
135         if err != nil {
136             return nil, 0, err
137         }
138     }
139     if p < 0 || p > 0xFFFF {
140         return nil, 0, &AddrError{"invalid port", po
141     }
142
143     return addr, p, nil

```

144
145 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/ipsock_posix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 package net
8
9 import "syscall"
10
11 // Should we try to use the IPv4 socket interface if we're
12 // only dealing with IPv4 sockets? As long as the host syst
13 // understands IPv6, it's okay to pass IPv4 addresses to the
14 // interface. That simplifies our code and is most general.
15 // Unfortunately, we need to run on kernels built without IP
16 // support too. So probe the kernel to figure it out.
17 //
18 // probeIPv6Stack probes both basic IPv6 capability and IPv6
19 // mapping capability which is controlled by IPV6_V6ONLY soc
20 // option and/or kernel state "net.inet6.ip6.v6only".
21 // It returns two boolean values. If the first boolean valu
22 // true, kernel supports basic IPv6 functionality. If the s
23 // boolean value is true, kernel supports IPV6 IPv4-mapping.
24 func probeIPv6Stack() (supportsIPv6, supportsIPv4map bool) {
25     var probes = []struct {
26         la TCPAddr
27         ok bool
28     }{
29         // IPv6 communication capability
30         {TCPAddr{IP: ParseIP("::1")}, false},
31         // IPv6 IPv4-mapped address communication ca
32         {TCPAddr{IP: IPv4(127, 0, 0, 1)}, false},
33     }
34
35     for i := range probes {
36         s, err := syscall.Socket(syscall.AF_INET6, s
37         if err != nil {
38             continue
39         }
40         defer closesocket(s)
41         syscall.SetsockoptInt(s, syscall.IPPROTO_IPV
```

```

42         sa, err := probes[i].la.toAddr().sockaddr(sy
43         if err != nil {
44             continue
45         }
46         err = syscall.Bind(s, sa)
47         if err != nil {
48             continue
49         }
50         probes[i].ok = true
51     }
52
53     return probes[0].ok, probes[1].ok
54 }
55
56 // favoriteAddrFamily returns the appropriate address family
57 // the given net, laddr, raddr and mode. At first it figure
58 // address family out from the net. If mode indicates "list
59 // and laddr is a wildcard, it assumes that the user wants t
60 // make a passive connection with a wildcard address family,
61 // AF_INET and AF_INET6, and a wildcard address like followi
62 //
63 //     1. A wild-wild listen, "tcp" + ""
64 //     If the platform supports both IPv6 and IPv4 IPv4-map
65 //     capabilities, we assume that the user want to listen
66 //     both IPv4 and IPv6 wildcard address over an AF_INET6
67 //     socket with IPV6_V6ONLY=0. Otherwise we prefer an I
68 //     wildcard address listen over an AF_INET socket.
69 //
70 //     2. A wild-ipv4wild listen, "tcp" + "0.0.0.0"
71 //     Same as 1.
72 //
73 //     3. A wild-ipv6wild listen, "tcp" + "[::]"
74 //     Almost same as 1 but we prefer an IPv6 wildcard addr
75 //     listen over an AF_INET6 socket with IPV6_V6ONLY=0 wh
76 //     the platform supports IPv6 capability but not IPv6 I
77 //     mapping capability.
78 //
79 //     4. A ipv4-ipv4wild listen, "tcp4" + "" or "0.0.0.0"
80 //     We use an IPv4 (AF_INET) wildcard address listen.
81 //
82 //     5. A ipv6-ipv6wild listen, "tcp6" + "" or "[::]"
83 //     We use an IPv6 (AF_INET6, IPV6_V6ONLY=1) wildcard ad
84 //     listen.
85 //
86 // Otherwise guess: if the addresses are IPv4 then returns A
87 // or else returns AF_INET6. It also returns a boolean valu
88 // designates IPV6_V6ONLY option.
89 //
90 // Note that OpenBSD allows neither "net.inet6.ip6.v6only=1"
91 // nor IPPROTO_IPV6 level IPV6_V6ONLY socket option setting.

```

```

92 func favoriteAddrFamily(net string, laddr, raddr sockaddr, r
93     switch net[len(net)-1] {
94     case '4':
95         return syscall.AF_INET, false
96     case '6':
97         return syscall.AF_INET6, true
98     }
99
100     if mode == "listen" && laddr.isWildcard() {
101         if supportsIPv4map {
102             return syscall.AF_INET6, false
103         }
104         return laddr.family(), false
105     }
106
107     if (laddr == nil || laddr.family() == syscall.AF_INET
108         (raddr == nil || raddr.family() == syscall.AF_INET6)
109         return syscall.AF_INET, false
110     }
111     return syscall.AF_INET6, false
112 }
113
114 // Internet sockets (TCP, UDP, IP)
115
116 // A sockaddr represents a TCP, UDP or IP network address that
117 // be converted into a syscall.Sockaddr.
118 type sockaddr interface {
119     Addr
120     family() int
121     isWildcard() bool
122     sockaddr(family int) (syscall.Sockaddr, error)
123 }
124
125 func internetSocket(net string, laddr, raddr sockaddr, sotyp
126     var la, ra syscall.Sockaddr
127     family, ipv6only := favoriteAddrFamily(net, laddr, r
128     if laddr != nil {
129         if la, err = laddr.sockaddr(family); err !=
130             goto Error
131     }
132
133     if raddr != nil {
134         if ra, err = raddr.sockaddr(family); err !=
135             goto Error
136     }
137
138     fd, err = socket(net, family, sotype, proto, ipv6only)
139     if err != nil {
140         goto Error

```

```

141     }
142     return fd, nil
143
144 Error:
145     addr := raddr
146     if mode == "listen" {
147         addr = laddr
148     }
149     return nil, &OpError{mode, net, addr, err}
150 }
151
152 func ipToSockaddr(family int, ip IP, port int) (syscall.Sock
153     switch family {
154     case syscall.AF_INET:
155         if len(ip) == 0 {
156             ip = IPv4zero
157         }
158         if ip = ip.To4(); ip == nil {
159             return nil, InvalidAddrError("non-IP
160         }
161         s := new(syscall.SockaddrInet4)
162         for i := 0; i < IPv4len; i++ {
163             s.Addr[i] = ip[i]
164         }
165         s.Port = port
166         return s, nil
167     case syscall.AF_INET6:
168         if len(ip) == 0 {
169             ip = IPv6zero
170         }
171         // IPv4 callers use 0.0.0.0 to mean "announc
172         // In IPv6 mode, Linux treats that as meanin
173         // which it refuses to do. Rewrite to the I
174         if ip.Equal(IPv4zero) {
175             ip = IPv6zero
176         }
177         if ip = ip.To16(); ip == nil {
178             return nil, InvalidAddrError("non-IP
179         }
180         s := new(syscall.SockaddrInet6)
181         for i := 0; i < IPv6len; i++ {
182             s.Addr[i] = ip[i]
183         }
184         s.Port = port
185         return s, nil
186     }
187     return nil, InvalidAddrError("unexpected socket fami
188 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/lookup_unix.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package net
8
9 import (
10     "errors"
11     "sync"
12 )
13
14 var (
15     protocols      map[string]int
16     onceReadProtocols sync.Once
17 )
18
19 // readProtocols loads contents of /etc/protocols into proto
20 // for quick access.
21 func readProtocols() {
22     protocols = make(map[string]int)
23     if file, err := open("/etc/protocols"); err == nil {
24         for line, ok := file.readLine(); ok; line, c
25             // tcp    6    TCP    # transmission
26             if i := byteIndex(line, '#'); i >= 0
27                 line = line[0:i]
28         }
29         f := getFields(line)
30         if len(f) < 2 {
31             continue
32         }
33         if proto, _, ok := dttoi(f[1], 0); ok
34             protocols[f[0]] = proto
35             for _, alias := range f[2:]
36                 protocols[alias] = p
37         }
38     }
39     file.close()
40 }
41 }
```

```

42 }
43
44 // lookupProtocol looks up IP protocol name in /etc/protocol
45 // returns correspondent protocol number.
46 func lookupProtocol(name string) (proto int, err error) {
47     onceReadProtocols.Do(readProtocols)
48     proto, found := protocols[name]
49     if !found {
50         return 0, errors.New("unknown IP protocol sp
51     }
52     return
53 }
54
55 func lookupHost(host string) (addrs []string, err error) {
56     addrs, err, ok := cgoLookupHost(host)
57     if !ok {
58         addrs, err = goLookupHost(host)
59     }
60     return
61 }
62
63 func lookupIP(host string) (addrs []IP, err error) {
64     addrs, err, ok := cgoLookupIP(host)
65     if !ok {
66         addrs, err = goLookupIP(host)
67     }
68     return
69 }
70
71 func lookupPort(network, service string) (port int, err error) {
72     port, err, ok := cgoLookupPort(network, service)
73     if !ok {
74         port, err = goLookupPort(network, service)
75     }
76     return
77 }
78
79 func lookupCNAME(name string) (cname string, err error) {
80     cname, err, ok := cgoLookupCNAME(name)
81     if !ok {
82         cname, err = goLookupCNAME(name)
83     }
84     return
85 }
86
87 func lookupSRV(service, proto, name string) (cname string, a
88     var target string
89     if service == "" && proto == "" {
90         target = name
91     } else {

```

```

92         target = "_" + service + "._" + proto + "."
93     }
94     var records []dnsRR
95     cname, records, err = lookup(target, dnsTypeSRV)
96     if err != nil {
97         return
98     }
99     addrs = make([]*SRV, len(records))
100    for i, rr := range records {
101        r := rr.(*dnsRR_SRV)
102        addrs[i] = &SRV{r.Target, r.Port, r.Priority}
103    }
104    byPriorityWeight(addrs).sort()
105    return
106 }
107
108 func lookupMX(name string) (mx []*MX, err error) {
109     _, records, err := lookup(name, dnsTypeMX)
110     if err != nil {
111         return
112     }
113     mx = make([]*MX, len(records))
114     for i, rr := range records {
115         r := rr.(*dnsRR_MX)
116         mx[i] = &MX{r.Mx, r.Pref}
117     }
118     byPref(mx).sort()
119     return
120 }
121
122 func lookupTXT(name string) (txt []string, err error) {
123     _, records, err := lookup(name, dnsTypeTXT)
124     if err != nil {
125         return
126     }
127     txt = make([]string, len(records))
128     for i, r := range records {
129         txt[i] = r.(*dnsRR_TXT).Txt
130     }
131     return
132 }
133
134 func lookupAddr(addr string) (name []string, err error) {
135     name = lookupStaticAddr(addr)
136     if len(name) > 0 {
137         return
138     }
139     var arpa string
140     arpa, err = reverseaddr(addr)

```

```
141     if err != nil {
142         return
143     }
144     var records []dnsRR
145     _, records, err = lookup(arpa, dnsTypePTR)
146     if err != nil {
147         return
148     }
149     name = make([]string, len(records))
150     for i := range records {
151         r := records[i].(*dnsRR_PTR)
152         name[i] = r.Ptr
153     }
154     return
155 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/mac.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // MAC address manipulations
6
7 package net
8
9 import "errors"
10
11 const hexDigit = "0123456789abcdef"
12
13 // A HardwareAddr represents a physical hardware address.
14 type HardwareAddr []byte
15
16 func (a HardwareAddr) String() string {
17     if len(a) == 0 {
18         return ""
19     }
20     buf := make([]byte, 0, len(a)*3-1)
21     for i, b := range a {
22         if i > 0 {
23             buf = append(buf, ':')
24         }
25         buf = append(buf, hexDigit[b>>4])
26         buf = append(buf, hexDigit[b&0xF])
27     }
28     return string(buf)
29 }
30
31 // ParseMAC parses s as an IEEE 802 MAC-48, EUI-48, or EUI-6
32 // following formats:
33 // 01:23:45:67:89:ab
34 // 01:23:45:67:89:ab:cd:ef
35 // 01-23-45-67-89-ab
36 // 01-23-45-67-89-ab-cd-ef
37 // 0123.4567.89ab
38 // 0123.4567.89ab.cdef
39 func ParseMAC(s string) (hw HardwareAddr, err error) {
40     if len(s) < 14 {
41         goto error
42     }
43
44     if s[2] == ':' || s[2] == '-' {
```

```

45         if (len(s)+1)%3 != 0 {
46             goto error
47         }
48         n := (len(s) + 1) / 3
49         if n != 6 && n != 8 {
50             goto error
51         }
52         hw = make(HardwareAddr, n)
53         for x, i := 0, 0; i < n; i++ {
54             var ok bool
55             if hw[i], ok = xtoi2(s[x:], s[2]); !
56                 goto error
57             }
58             x += 3
59         }
60     } else if s[4] == '.' {
61         if (len(s)+1)%5 != 0 {
62             goto error
63         }
64         n := 2 * (len(s) + 1) / 5
65         if n != 6 && n != 8 {
66             goto error
67         }
68         hw = make(HardwareAddr, n)
69         for x, i := 0, 0; i < n; i += 2 {
70             var ok bool
71             if hw[i], ok = xtoi2(s[x:x+2], 0); !
72                 goto error
73             }
74             if hw[i+1], ok = xtoi2(s[x+2:], s[4])
75                 goto error
76             }
77             x += 5
78         }
79     } else {
80         goto error
81     }
82     return hw, nil
83
84 error:
85     return nil, errors.New("invalid MAC address: " + s)
86 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/net.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6 Package net provides a portable interface for network I/O, i
7 TCP/IP, UDP, domain name resolution, and Unix domain sockets
8
9 Although the package provides access to low-level networking
10 primitives, most clients will need only the basic interface
11 by the Dial, Listen, and Accept functions and the associated
12 Conn and Listener interfaces. The crypto/tls package uses
13 the same interfaces and similar Dial and Listen functions.
14
15 The Dial function connects to a server:
16
17     conn, err := net.Dial("tcp", "google.com:80")
18     if err != nil {
19         // handle error
20     }
21     fmt.Fprintf(conn, "GET / HTTP/1.0\r\n\r\n")
22     status, err := bufio.NewReader(conn).ReadString('\n')
23     // ...
24
25 The Listen function creates servers:
26
27     ln, err := net.Listen("tcp", ":8080")
28     if err != nil {
29         // handle error
30     }
31     for {
32         conn, err := ln.Accept()
33         if err != nil {
34             // handle error
35             continue
36         }
37         go handleConnection(conn)
38     }
39 */
40 package net
41
42 // TODO(rsc):
43 //     support for raw ethernet sockets
44
```

```

45 import (
46     "errors"
47     "time"
48 )
49
50 // Addr represents a network end point address.
51 type Addr interface {
52     Network() string // name of the network
53     String() string  // string form of address
54 }
55
56 // Conn is a generic stream-oriented network connection.
57 //
58 // Multiple goroutines may invoke methods on a Conn simultan
59 type Conn interface {
60     // Read reads data from the connection.
61     // Read can be made to time out and return a Error w
62     // after a fixed time limit; see SetDeadline and Set
63     Read(b []byte) (n int, err error)
64
65     // Write writes data to the connection.
66     // Write can be made to time out and return a Error
67     // after a fixed time limit; see SetDeadline and Set
68     Write(b []byte) (n int, err error)
69
70     // Close closes the connection.
71     // Any blocked Read or Write operations will be unbl
72     Close() error
73
74     // LocalAddr returns the local network address.
75     LocalAddr() Addr
76
77     // RemoteAddr returns the remote network address.
78     RemoteAddr() Addr
79
80     // SetDeadline sets the read and write deadlines ass
81     // with the connection. It is equivalent to calling
82     // SetReadDeadline and SetWriteDeadline.
83     //
84     // A deadline is an absolute time after which I/O op
85     // fail with a timeout (see type Error) instead of
86     // blocking. The deadline applies to all future I/O,
87     // the immediately following call to Read or Write.
88     //
89     // An idle timeout can be implemented by repeatedly
90     // the deadline after successful Read or Write calls
91     //
92     // A zero value for t means I/O operations will not
93     SetDeadline(t time.Time) error
94

```

```

95         // SetReadDeadline sets the deadline for future Read
96         // A zero value for t means Read will not time out.
97         SetReadDeadline(t time.Time) error
98
99         // SetWriteDeadline sets the deadline for future Write
100        // Even if write times out, it may return n > 0, indicating
101        // some of the data was successfully written.
102        // A zero value for t means Write will not time out.
103        SetWriteDeadline(t time.Time) error
104    }
105
106    // An Error represents a network error.
107    type Error interface {
108        error
109        Timeout() bool // Is the error a timeout?
110        Temporary() bool // Is the error temporary?
111    }
112
113    // PacketConn is a generic packet-oriented network connection.
114    //
115    // Multiple goroutines may invoke methods on a PacketConn simultaneously.
116    type PacketConn interface {
117        // ReadFrom reads a packet from the connection,
118        // copying the payload into b. It returns the number of
119        // bytes copied into b and the return address that
120        // was on the packet.
121        // ReadFrom can be made to time out and return
122        // an error with Timeout() == true after a fixed timeout period.
123        // see SetDeadline and SetReadDeadline.
124        ReadFrom(b []byte) (n int, addr Addr, err error)
125
126        // WriteTo writes a packet with payload b to addr.
127        // WriteTo can be made to time out and return
128        // an error with Timeout() == true after a fixed timeout period.
129        // see SetDeadline and SetWriteDeadline.
130        // On packet-oriented connections, write timeouts are not
131        // supported.
132        WriteTo(b []byte, addr Addr) (n int, err error)
133
134        // Close closes the connection.
135        // Any blocked ReadFrom or WriteTo operations will be unblocked.
136        Close() error
137
138        // LocalAddr returns the local network address.
139        LocalAddr() Addr
140
141        // SetDeadline sets the read and write deadlines associated
142        // with the connection.
143        SetDeadline(t time.Time) error

```

```

144         // SetReadDeadline sets the deadline for future Read
145         // If the deadline is reached, Read will fail with a
146         // (see type Error) instead of blocking.
147         // A zero value for t means Read will not time out.
148         SetReadDeadline(t time.Time) error
149
150         // SetWriteDeadline sets the deadline for future Wri
151         // If the deadline is reached, Write will fail with
152         // (see type Error) instead of blocking.
153         // A zero value for t means Write will not time out.
154         // Even if write times out, it may return n > 0, ind
155         // some of the data was successfully written.
156         SetWriteDeadline(t time.Time) error
157     }
158
159     // A Listener is a generic network listener for stream-orien
160     //
161     // Multiple goroutines may invoke methods on a Listener simu
162     type Listener interface {
163         // Accept waits for and returns the next connection
164         Accept() (c Conn, err error)
165
166         // Close closes the listener.
167         // Any blocked Accept operations will be unblocked a
168         Close() error
169
170         // Addr returns the listener's network address.
171         Addr() Addr
172     }
173
174     var errMissingAddress = errors.New("missing address")
175
176     type OpError struct {
177         Op    string
178         Net   string
179         Addr  Addr
180         Err   error
181     }
182
183     func (e *OpError) Error() string {
184         if e == nil {
185             return "<nil>"
186         }
187         s := e.Op
188         if e.Net != "" {
189             s += " " + e.Net
190         }
191         if e.Addr != nil {
192             s += " " + e.Addr.String()

```

```

193     }
194     s += ": " + e.Err.Error()
195     return s
196 }
197
198 type temporary interface {
199     Temporary() bool
200 }
201
202 func (e *OpError) Temporary() bool {
203     t, ok := e.Err.(temporary)
204     return ok && t.Temporary()
205 }
206
207 type timeout interface {
208     Timeout() bool
209 }
210
211 func (e *OpError) Timeout() bool {
212     t, ok := e.Err.(timeout)
213     return ok && t.Timeout()
214 }
215
216 type timeoutError struct{}
217
218 func (e *timeoutError) Error() string { return "i/o timeou"
219 func (e *timeoutError) Timeout() bool { return true }
220 func (e *timeoutError) Temporary() bool { return true }
221
222 var errTimeout error = &timeoutError{}
223
224 type AddrError struct {
225     Err string
226     Addr string
227 }
228
229 func (e *AddrError) Error() string {
230     if e == nil {
231         return "<nil>"
232     }
233     s := e.Err
234     if e.Addr != "" {
235         s += " " + e.Addr
236     }
237     return s
238 }
239
240 func (e *AddrError) Temporary() bool {
241     return false
242 }

```

```
243
244 func (e *AddrError) Timeout() bool {
245     return false
246 }
247
248 type UnknownNetworkError string
249
250 func (e UnknownNetworkError) Error() string { return "unkn
251 func (e UnknownNetworkError) Temporary() bool { return false
252 func (e UnknownNetworkError) Timeout() bool { return false
253
254 // DNSConfigError represents an error reading the machine's
255 type DNSConfigError struct {
256     Err error
257 }
258
259 func (e *DNSConfigError) Error() string {
260     return "error reading DNS config: " + e.Err.Error()
261 }
262
263 func (e *DNSConfigError) Timeout() bool { return false }
264 func (e *DNSConfigError) Temporary() bool { return false }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/newpollserver.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package net
8
9 import (
10     "os"
11     "syscall"
12 )
13
14 func newPollServer() (s *pollServer, err error) {
15     s = new(pollServer)
16     s.cr = make(chan *netFD, 1)
17     s.cw = make(chan *netFD, 1)
18     if s.pr, s.pw, err = os.Pipe(); err != nil {
19         return nil, err
20     }
21     if err = syscall.SetNonblock(int(s.pr.Fd()), true);
22         goto Errno
23     }
24     if err = syscall.SetNonblock(int(s.pw.Fd()), true);
25         goto Errno
26     }
27     if s.poll, err = newpollster(); err != nil {
28         goto Error
29     }
30     if _, err = s.poll.AddFD(int(s.pr.Fd()), 'r', true);
31         s.poll.Close()
32         goto Error
33     }
34     s.pending = make(map[int]*netFD)
35     go s.Run()
36     return s, nil
37
38 Errno:
39     err = &os.PathError{
40         Op:   "setnonblock",
41         Path: s.pr.Name(),
```

```
42             Err:  err,  
43         }  
44     Error:  
45         s.pr.Close()  
46         s.pw.Close()  
47         return nil, err  
48     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/parse.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Simple file i/o and string manipulation, to avoid
6 // depending on strconv and bufio and strings.
7
8 package net
9
10 import (
11     "io"
12     "os"
13 )
14
15 type file struct {
16     file *os.File
17     data []byte
18     atEOF bool
19 }
20
21 func (f *file) close() { f.file.Close() }
22
23 func (f *file) getLineFromData() (s string, ok bool) {
24     data := f.data
25     i := 0
26     for i = 0; i < len(data); i++ {
27         if data[i] == '\n' {
28             s = string(data[0:i])
29             ok = true
30             // move data
31             i++
32             n := len(data) - i
33             copy(data[0:], data[i:])
34             f.data = data[0:n]
35             return
36         }
37     }
38     if f.atEOF && len(f.data) > 0 {
39         // EOF, return all we have
40         s = string(data)
41         f.data = f.data[0:0]
42         ok = true
43     }
44     return
```

```

45 }
46
47 func (f *file) readLine() (s string, ok bool) {
48     if s, ok = f.getLineFromData(); ok {
49         return
50     }
51     if len(f.data) < cap(f.data) {
52         ln := len(f.data)
53         n, err := io.ReadFull(f.file, f.data[ln:cap(
54             if n >= 0 {
55                 f.data = f.data[0 : ln+n]
56             }
57             if err == io.EOF {
58                 f.atEOF = true
59             }
60         }
61         s, ok = f.getLineFromData()
62         return
63     }
64
65 func open(name string) (*file, error) {
66     fd, err := os.Open(name)
67     if err != nil {
68         return nil, err
69     }
70     return &file{fd, make([]byte, os.Getpagesize())[0:0]}
71 }
72
73 func byteIndex(s string, c byte) int {
74     for i := 0; i < len(s); i++ {
75         if s[i] == c {
76             return i
77         }
78     }
79     return -1
80 }
81
82 // Count occurrences in s of any bytes in t.
83 func countAnyByte(s string, t string) int {
84     n := 0
85     for i := 0; i < len(s); i++ {
86         if byteIndex(t, s[i]) >= 0 {
87             n++
88         }
89     }
90     return n
91 }
92
93 // Split s at any bytes in t.
94 func splitAtBytes(s string, t string) []string {

```

```

95     a := make([]string, 1+countAnyByte(s, t))
96     n := 0
97     last := 0
98     for i := 0; i < len(s); i++ {
99         if byteIndex(t, s[i]) >= 0 {
100             if last < i {
101                 a[n] = string(s[last:i])
102                 n++
103             }
104             last = i + 1
105         }
106     }
107     if last < len(s) {
108         a[n] = string(s[last:])
109         n++
110     }
111     return a[0:n]
112 }
113
114 func getFields(s string) []string { return splitAtBytes(s, "
115
116 // Bigger than we need, not too big to worry about overflow
117 const big = 0xFFFFF
118
119 // Decimal to integer starting at &s[i0].
120 // Returns number, new offset, success.
121 func dtoi(s string, i0 int) (n int, i int, ok bool) {
122     n = 0
123     for i = i0; i < len(s) && '0' <= s[i] && s[i] <= '9'
124         n = n*10 + int(s[i]-'0')
125         if n >= big {
126             return 0, i, false
127         }
128     }
129     if i == i0 {
130         return 0, i, false
131     }
132     return n, i, true
133 }
134
135 // Hexadecimal to integer starting at &s[i0].
136 // Returns number, new offset, success.
137 func xtoi(s string, i0 int) (n int, i int, ok bool) {
138     n = 0
139     for i = i0; i < len(s); i++ {
140         if '0' <= s[i] && s[i] <= '9' {
141             n *= 16
142             n += int(s[i] - '0')
143         } else if 'a' <= s[i] && s[i] <= 'f' {

```

```

144             n *= 16
145             n += int(s[i]-'a') + 10
146         } else if 'A' <= s[i] && s[i] <= 'F' {
147             n *= 16
148             n += int(s[i]-'A') + 10
149         } else {
150             break
151         }
152         if n >= big {
153             return 0, i, false
154         }
155     }
156     if i == i0 {
157         return 0, i, false
158     }
159     return n, i, true
160 }
161
162 // xtoi2 converts the next two hex digits of s into a byte.
163 // If s is longer than 2 bytes then the third byte must be e
164 // If the first two bytes of s are not hex digits or the thi
165 // does not match e, false is returned.
166 func xtoi2(s string, e byte) (byte, bool) {
167     if len(s) > 2 && s[2] != e {
168         return 0, false
169     }
170     n, ei, ok := xtoi(s[:2], 0)
171     return byte(n), ok && ei == 2
172 }
173
174 // Integer to decimal.
175 func itoa(i int) string {
176     var buf [30]byte
177     n := len(buf)
178     neg := false
179     if i < 0 {
180         i = -i
181         neg = true
182     }
183     ui := uint(i)
184     for ui > 0 || n == len(buf) {
185         n--
186         buf[n] = byte('0' + ui%10)
187         ui /= 10
188     }
189     if neg {
190         n--
191         buf[n] = '-'
192     }

```

```

193         return string(buf[n:])
194     }
195
196 // Convert i to decimal string.
197 func itod(i uint) string {
198     if i == 0 {
199         return "0"
200     }
201
202     // Assemble decimal in reverse order.
203     var b [32]byte
204     bp := len(b)
205     for ; i > 0; i /= 10 {
206         bp--
207         b[bp] = byte(i%10) + '0'
208     }
209
210     return string(b[bp:])
211 }
212
213 // Convert i to hexadecimal string.
214 func itox(i uint, min int) string {
215     // Assemble hexadecimal in reverse order.
216     var b [32]byte
217     bp := len(b)
218     for ; i > 0 || min > 0; i /= 16 {
219         bp--
220         b[bp] = "0123456789abcdef"[byte(i%16)]
221         min--
222     }
223
224     return string(b[bp:])
225 }
226
227 // Number of occurrences of b in s.
228 func count(s string, b byte) int {
229     n := 0
230     for i := 0; i < len(s); i++ {
231         if s[i] == b {
232             n++
233         }
234     }
235     return n
236 }
237
238 // Index of rightmost occurrence of b in s.
239 func last(s string, b byte) int {
240     i := len(s)
241     for i--; i >= 0; i-- {
242         if s[i] == b {

```

```
243             break
244         }
245     }
246     return i
247 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/pipe.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package net
6
7 import (
8     "errors"
9     "io"
10    "time"
11 )
12
13 // Pipe creates a synchronous, in-memory, full duplex
14 // network connection; both ends implement the Conn interface.
15 // Reads on one end are matched with writes on the other,
16 // copying data directly between the two; there is no internal
17 // buffering.
18 func Pipe() (Conn, Conn) {
19     r1, w1 := io.Pipe()
20     r2, w2 := io.Pipe()
21
22     return &pipe{r1, w2}, &pipe{r2, w1}
23 }
24
25 type pipe struct {
26     *io.PipeReader
27     *io.PipeWriter
28 }
29
30 type pipeAddr int
31
32 func (pipeAddr) Network() string {
33     return "pipe"
34 }
35
36 func (pipeAddr) String() string {
37     return "pipe"
38 }
39
40 func (p *pipe) Close() error {
41     err := p.PipeReader.Close()
42     err1 := p.PipeWriter.Close()
43     if err == nil {
44         err = err1
45     }
46 }
```

```
45         }
46         return err
47     }
48
49     func (p *pipe) LocalAddr() Addr {
50         return pipeAddr(0)
51     }
52
53     func (p *pipe) RemoteAddr() Addr {
54         return pipeAddr(0)
55     }
56
57     func (p *pipe) SetDeadline(t time.Time) error {
58         return errors.New("net.Pipe does not support deadlin
59     }
60
61     func (p *pipe) SetReadDeadline(t time.Time) error {
62         return errors.New("net.Pipe does not support deadlin
63     }
64
65     func (p *pipe) SetWriteDeadline(t time.Time) error {
66         return errors.New("net.Pipe does not support deadlin
67     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/port.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 // Read system port mappings from /etc/services
8
9 package net
10
11 import "sync"
12
13 var services map[string]map[string]int
14 var servicesError error
15 var onceReadServices sync.Once
16
17 func readServices() {
18     services = make(map[string]map[string]int)
19     var file *file
20     if file, servicesError = open("/etc/services"); serv
21         return
22     }
23     for line, ok := file.readLine(); ok; line, ok = file
24         // "http 80/tcp www www-http # World Wide We
25         if i := byteIndex(line, '#'); i >= 0 {
26             line = line[0:i]
27         }
28         f := getFields(line)
29         if len(f) < 2 {
30             continue
31         }
32         portnet := f[1] // "tcp/80"
33         port, j, ok := dtol(portnet, 0)
34         if !ok || port <= 0 || j >= len(portnet) ||
35             continue
36     }
37     netw := portnet[j+1:] // "tcp"
38     m, ok1 := services[netw]
39     if !ok1 {
40         m = make(map[string]int)
41         services[netw] = m
42     }
43     for i := 0; i < len(f); i++ {
44         if i != 1 { // f[1] was port/net
```

```

45             m[f[i]] = port
46         }
47     }
48 }
49     file.close()
50 }
51
52 // goLookupPort is the native Go implementation of LookupPort
53 func goLookupPort(network, service string) (port int, err error) {
54     onceReadServices.Do(readServices)
55
56     switch network {
57     case "tcp4", "tcp6":
58         network = "tcp"
59     case "udp4", "udp6":
60         network = "udp"
61     }
62
63     if m, ok := services[network]; ok {
64         if port, ok = m[service]; ok {
65             return
66         }
67     }
68     return 0, &AddrError{"unknown port", network + "/" +
69 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/sendfile_linux.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package net
6
7 import (
8     "io"
9     "os"
10    "syscall"
11 )
12
13 // maxSendfileSize is the largest chunk size we ask the kern
14 // at a time.
15 const maxSendfileSize int = 4 << 20
16
17 // sendFile copies the contents of r to c using the sendfile
18 // system call to minimize copies.
19 //
20 // if handled == true, sendFile returns the number of bytes
21 // non-EOF error.
22 //
23 // if handled == false, sendFile performed no work.
24 func sendFile(c *netFD, r io.Reader) (written int64, err error) {
25     var remain int64 = 1 << 62 // by default, copy until
26
27     lr, ok := r.(*io.LimitedReader)
28     if ok {
29         remain, r = lr.N, lr.R
30         if remain <= 0 {
31             return 0, nil, true
32         }
33     }
34     f, ok := r.(*os.File)
35     if !ok {
36         return 0, nil, false
37     }
38
39     c.wio.Lock()
40     defer c.wio.Unlock()
41     if err := c.incref(false); err != nil {
```

```

42         return 0, err, true
43     }
44     defer c.decref()
45
46     dst := c.sysfd
47     src := int(f.Fd())
48     for remain > 0 {
49         n := maxSendfileSize
50         if int64(n) > remain {
51             n = int(remain)
52         }
53         n, err1 := syscall.Sendfile(dst, src, nil, n)
54         if n > 0 {
55             written += int64(n)
56             remain -= int64(n)
57         }
58         if n == 0 && err1 == nil {
59             break
60         }
61         if err1 == syscall.EAGAIN && c.wdeadline >=
62             if err1 = pollserver.WaitWrite(c); e
63                 continue
64             }
65         }
66         if err1 != nil {
67             // This includes syscall.ENOSYS (no
68             // support) and syscall.EINVAL (fd t
69             // don't implement sendfile together
70             err = &OpError{"sendfile", c.net, c.
71             break
72         }
73     }
74     if lr != nil {
75         lr.N = remain
76     }
77     return written, err, written > 0
78 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/sock.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 // Sockets
8
9 package net
10
11 import (
12     "io"
13     "syscall"
14 )
15
16 var listenerBacklog = maxListenerBacklog()
17
18 // Generic socket creation.
19 func socket(net string, f, t, p int, ipv6only bool, la, ra s
20     // See ../syscall/exec.go for description of ForkLoc
21     syscall.ForkLock.RLock()
22     s, err := syscall.Socket(f, t, p)
23     if err != nil {
24         syscall.ForkLock.RUnlock()
25         return nil, err
26     }
27     syscall.CloseOnExec(s)
28     syscall.ForkLock.RUnlock()
29
30     err = setDefaultSockopts(s, f, t, ipv6only)
31     if err != nil {
32         closesocket(s)
33         return nil, err
34     }
35
36     var bla syscall.Sockaddr
37     if la != nil {
38         bla, err = listenerSockaddr(s, f, la, toAddr
39         if err != nil {
40             closesocket(s)
41             return nil, err
42         }
43         err = syscall.Bind(s, bla)
44         if err != nil {
```

```

45             closesocket(s)
46             return nil, err
47         }
48     }
49
50     if fd, err = newFD(s, f, t, net); err != nil {
51         closesocket(s)
52         return nil, err
53     }
54
55     if ra != nil {
56         if err = fd.connect(ra); err != nil {
57             closesocket(s)
58             fd.Close()
59             return nil, err
60         }
61         fd.isConnected = true
62     }
63
64     sa, _ := syscall.Getsockname(s)
65     var laddr Addr
66     if la != nil && bla != la {
67         laddr = toAddr(la)
68     } else {
69         laddr = toAddr(sa)
70     }
71     sa, _ = syscall.Getpeername(s)
72     raddr := toAddr(sa)
73
74     fd.setAddr(laddr, raddr)
75     return fd, nil
76 }
77
78 type writerOnly struct {
79     io.Writer
80 }
81
82 // Fallback implementation of io.ReaderFrom's ReadFrom, when
83 // applicable.
84 func genericReadFrom(w io.Writer, r io.Reader) (n int64, err
85     // Use wrapper to hide existing r.ReadFrom from io.C
86     return io.Copy(writerOnly{w}, r)
87 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/sock_linux.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Sockets for Linux
6
7 package net
8
9 import "syscall"
10
11 func maxListenerBacklog() int {
12     fd, err := open("/proc/sys/net/core/somaxconn")
13     if err != nil {
14         return syscall.SOMAXCONN
15     }
16     defer fd.close()
17     l, ok := fd.readLine()
18     if !ok {
19         return syscall.SOMAXCONN
20     }
21     f := getFields(l)
22     n, _, ok := dtoi(f[0], 0)
23     if n == 0 || !ok {
24         return syscall.SOMAXCONN
25     }
26     return n
27 }
28
29 func listenerSockaddr(s, f int, la syscall.Sockaddr, toAddr
30     a := toAddr(la)
31     if a == nil {
32         return la, nil
33     }
34     switch v := a.(type) {
35     case *TCPAddr, *UnixAddr:
36         err := setDefaultListenerSockopts(s)
37         if err != nil {
38             return nil, err
39         }
40     case *UDPAddr:
41         if v.IP.IsMulticast() {
42             err := setDefaultMulticastSockopts(s)
43             if err != nil {
44                 return nil, err
```

```
45         }
46         switch f {
47         case syscall.AF_INET:
48             v.IP = IPv4zero
49         case syscall.AF_INET6:
50             v.IP = IPv6unspecified
51         }
52         return v.sockaddr(f)
53     }
54 }
55     return la, nil
56 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/sockopt.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 // Socket options
8
9 package net
10
11 import (
12     "os"
13     "syscall"
14     "time"
15 )
16
17 // Boolean to int.
18 func boolint(b bool) int {
19     if b {
20         return 1
21     }
22     return 0
23 }
24
25 func ipv4AddrToInterface(ip IP) (*Interface, error) {
26     ift, err := Interfaces()
27     if err != nil {
28         return nil, err
29     }
30     for _, ifi := range ift {
31         ifat, err := ifi.Addrs()
32         if err != nil {
33             return nil, err
34         }
35         for _, ifa := range ifat {
36             switch v := ifa.(type) {
37             case *IPAddr:
38                 if ip.Equal(v.IP) {
39                     return &ifi, nil
40                 }
41             case *IPNet:
42                 if ip.Equal(v.IP) {
43                     return &ifi, nil
44                 }
45             }
```

```

45         }
46     }
47 }
48 if ip.Equal(IPv4zero) {
49     return nil, nil
50 }
51 return nil, errNoSuchInterface
52 }
53
54 func interfaceToIPv4Addr(ifi *Interface) (IP, error) {
55     if ifi == nil {
56         return IPv4zero, nil
57     }
58     ifat, err := ifi.Addrs()
59     if err != nil {
60         return nil, err
61     }
62     for _, ifa := range ifat {
63         switch v := ifa.(type) {
64             case *IPAddr:
65                 if v.IP.To4() != nil {
66                     return v.IP, nil
67                 }
68             case *IPNet:
69                 if v.IP.To4() != nil {
70                     return v.IP, nil
71                 }
72         }
73     }
74     return nil, errNoSuchInterface
75 }
76
77 func setIPv4MreqToInterface(mreq *syscall.IPmreq, ifi *Inter
78     if ifi == nil {
79         return nil
80     }
81     ifat, err := ifi.Addrs()
82     if err != nil {
83         return err
84     }
85     for _, ifa := range ifat {
86         switch v := ifa.(type) {
87             case *IPAddr:
88                 if a := v.IP.To4(); a != nil {
89                     copy(mreq.Interface[:], a)
90                     goto done
91                 }
92             case *IPNet:
93                 if a := v.IP.To4(); a != nil {
94                     copy(mreq.Interface[:], a)

```

```

95             goto done
96         }
97     }
98 }
99 done:
100     if bytesEqual(mreq.Multiaddr[:], IPv4zero.To4()) {
101         return errNoSuchMulticastInterface
102     }
103     return nil
104 }
105
106 func setReadBuffer(fd *netFD, bytes int) error {
107     if err := fd.incref(false); err != nil {
108         return err
109     }
110     defer fd.decref()
111     return os.NewSyscallError("setsockopt", syscall.Sets
112 }
113
114 func setWriteBuffer(fd *netFD, bytes int) error {
115     if err := fd.incref(false); err != nil {
116         return err
117     }
118     defer fd.decref()
119     return os.NewSyscallError("setsockopt", syscall.Sets
120 }
121
122 func setReadDeadline(fd *netFD, t time.Time) error {
123     if t.IsZero() {
124         fd.rdeadline = 0
125     } else {
126         fd.rdeadline = t.UnixNano()
127     }
128     return nil
129 }
130
131 func setWriteDeadline(fd *netFD, t time.Time) error {
132     if t.IsZero() {
133         fd.wdeadline = 0
134     } else {
135         fd.wdeadline = t.UnixNano()
136     }
137     return nil
138 }
139
140 func setDeadline(fd *netFD, t time.Time) error {
141     if err := setReadDeadline(fd, t); err != nil {
142         return err
143     }

```

```

144         return setWriteDeadline(fd, t)
145     }
146
147     func setReuseAddr(fd *netFD, reuse bool) error {
148         if err := fd.incref(false); err != nil {
149             return err
150         }
151         defer fd.decref()
152         return os.NewSyscallError("setsockopt", syscall.Sets
153     }
154
155     func setDontRoute(fd *netFD, dontroute bool) error {
156         if err := fd.incref(false); err != nil {
157             return err
158         }
159         defer fd.decref()
160         return os.NewSyscallError("setsockopt", syscall.Sets
161     }
162
163     func setKeepAlive(fd *netFD, keepalive bool) error {
164         if err := fd.incref(false); err != nil {
165             return err
166         }
167         defer fd.decref()
168         return os.NewSyscallError("setsockopt", syscall.Sets
169     }
170
171     func setNoDelay(fd *netFD, noDelay bool) error {
172         if err := fd.incref(false); err != nil {
173             return err
174         }
175         defer fd.decref()
176         return os.NewSyscallError("setsockopt", syscall.Sets
177     }
178
179     func setLinger(fd *netFD, sec int) error {
180         var l syscall.Linger
181         if sec >= 0 {
182             l.Onoff = 1
183             l.Linger = int32(sec)
184         } else {
185             l.Onoff = 0
186             l.Linger = 0
187         }
188         if err := fd.incref(false); err != nil {
189             return err
190         }
191         defer fd.decref()
192         return os.NewSyscallError("setsockopt", syscall.Sets

```

193 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/sockopt_linux.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Socket options for Linux
6
7 package net
8
9 import (
10     "os"
11     "syscall"
12 )
13
14 func setDefaultSockopts(s, f, t int, ipv6only bool) error {
15     switch f {
16     case syscall.AF_INET6:
17         if ipv6only {
18             syscall.SetsockoptInt(s, syscall.IPP
19         } else {
20             // Allow both IP versions even if th
21             // is otherwise. Note that some ope
22             // never admit this option.
23             syscall.SetsockoptInt(s, syscall.IPP
24         }
25     }
26     // Allow broadcast.
27     err := syscall.SetsockoptInt(s, syscall.SOL_SOCKET,
28     if err != nil {
29         return os.NewSyscallError("setsockopt", err)
30     }
31     return nil
32 }
33
34 func setDefaultListenerSockopts(s int) error {
35     // Allow reuse of recently-used addresses.
36     err := syscall.SetsockoptInt(s, syscall.SOL_SOCKET,
37     if err != nil {
38         return os.NewSyscallError("setsockopt", err)
39     }
40     return nil
41 }
```

```
42
43 func setDefaultMulticastSockopts(s int) error {
44     // Allow multicast UDP and raw IP datagram sockets t
45     // concurrently across multiple listeners.
46     err := syscall.SetsockoptInt(s, syscall.SOL_SOCKET,
47     if err != nil {
48         return os.NewSyscallError("setsockopt", err)
49     }
50     return nil
51 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/sockoptip.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 // IP-level socket options
8
9 package net
10
11 import (
12     "os"
13     "syscall"
14 )
15
16 func ipv4TOS(fd *netFD) (int, error) {
17     if err := fd.incref(false); err != nil {
18         return 0, err
19     }
20     defer fd.decref()
21     v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IPPROTO_IP,
22     syscall.TOS)
23     if err != nil {
24         return 0, os.NewSyscallError("getsockopt", err)
25     }
26     return v, nil
27 }
28
29 func setIPv4TOS(fd *netFD, v int) error {
30     if err := fd.incref(false); err != nil {
31         return err
32     }
33     defer fd.decref()
34     err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPROTO_IP,
35     syscall.TOS, v)
36     if err != nil {
37         return os.NewSyscallError("setsockopt", err)
38     }
39     return nil
40 }
41
42 func ipv4TTL(fd *netFD) (int, error) {
43     if err := fd.incref(false); err != nil {
44         return 0, err
45     }
46     defer fd.decref()
47     v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IPPROTO_IP,
48     syscall.TTL)
49     if err != nil {
50         return 0, os.NewSyscallError("getsockopt", err)
51     }
52     return v, nil
53 }
54
55 func setIPv4TTL(fd *netFD, v int) error {
56     if err := fd.incref(false); err != nil {
57         return err
58     }
59     defer fd.decref()
60     err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPROTO_IP,
61     syscall.TTL, v)
62     if err != nil {
63         return os.NewSyscallError("setsockopt", err)
64     }
65     return nil
66 }
```

```

45         v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IP
46         if err != nil {
47             return 0, os.NewSyscallError("getsockopt", e
48         }
49         return v, nil
50     }
51
52     func setIPv4TTL(fd *netFD, v int) error {
53         if err := fd.incref(false); err != nil {
54             return err
55         }
56         defer fd.decref()
57         err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPRC
58         if err != nil {
59             return os.NewSyscallError("setsockopt", err)
60         }
61         return nil
62     }
63
64     func joinIPv4Group(fd *netFD, ifi *Interface, ip IP) error {
65         mreq := &syscall.IPmreq{Multiaddr: [4]byte{ip[0], ip
66         if err := setIPv4MreqToInterface(mreq, ifi); err !=
67             return err
68         }
69         if err := fd.incref(false); err != nil {
70             return err
71         }
72         defer fd.decref()
73         return os.NewSyscallError("setsockopt", syscall.Sets
74     }
75
76     func leaveIPv4Group(fd *netFD, ifi *Interface, ip IP) error
77         mreq := &syscall.IPmreq{Multiaddr: [4]byte{ip[0], ip
78         if err := setIPv4MreqToInterface(mreq, ifi); err !=
79             return err
80         }
81         if err := fd.incref(false); err != nil {
82             return err
83         }
84         defer fd.decref()
85         return os.NewSyscallError("setsockopt", syscall.Sets
86     }
87
88     func ipv6HopLimit(fd *netFD) (int, error) {
89         if err := fd.incref(false); err != nil {
90             return 0, err
91         }
92         defer fd.decref()
93         v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IP
94         if err != nil {

```

```

95         return 0, os.NewSyscallError("getsockopt", e
96     }
97     return v, nil
98 }
99
100 func setIPv6HopLimit(fd *netFD, v int) error {
101     if err := fd.incref(false); err != nil {
102         return err
103     }
104     defer fd.decref()
105     err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPRO
106     if err != nil {
107         return os.NewSyscallError("setsockopt", err)
108     }
109     return nil
110 }
111
112 func ipv6MulticastInterface(fd *netFD) (*Interface, error) {
113     if err := fd.incref(false); err != nil {
114         return nil, err
115     }
116     defer fd.decref()
117     v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IP
118     if err != nil {
119         return nil, os.NewSyscallError("getsockopt",
120     }
121     if v == 0 {
122         return nil, nil
123     }
124     ifi, err := InterfaceByIndex(v)
125     if err != nil {
126         return nil, err
127     }
128     return ifi, nil
129 }
130
131 func setIPv6MulticastInterface(fd *netFD, ifi *Interface) er
132     var v int
133     if ifi != nil {
134         v = ifi.Index
135     }
136     if err := fd.incref(false); err != nil {
137         return err
138     }
139     defer fd.decref()
140     err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPRO
141     if err != nil {
142         return os.NewSyscallError("setsockopt", err)
143     }

```

```

144         return nil
145     }
146
147     func ipv6MulticastHopLimit(fd *netFD) (int, error) {
148         if err := fd.incref(false); err != nil {
149             return 0, err
150         }
151         defer fd.decref()
152         v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IP
153         if err != nil {
154             return 0, os.NewSyscallError("getsockopt", e
155         }
156         return v, nil
157     }
158
159     func setIPv6MulticastHopLimit(fd *netFD, v int) error {
160         if err := fd.incref(false); err != nil {
161             return err
162         }
163         defer fd.decref()
164         err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPRO
165         if err != nil {
166             return os.NewSyscallError("setsockopt", err)
167         }
168         return nil
169     }
170
171     func ipv6MulticastLoopback(fd *netFD) (bool, error) {
172         if err := fd.incref(false); err != nil {
173             return false, err
174         }
175         defer fd.decref()
176         v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IP
177         if err != nil {
178             return false, os.NewSyscallError("getsockopt
179         }
180         return v == 1, nil
181     }
182
183     func setIPv6MulticastLoopback(fd *netFD, v bool) error {
184         if err := fd.incref(false); err != nil {
185             return err
186         }
187         defer fd.decref()
188         err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPRO
189         if err != nil {
190             return os.NewSyscallError("setsockopt", err)
191         }
192         return nil

```

```

193 }
194
195 func joinIPv6Group(fd *netFD, ifi *Interface, ip IP) error {
196     mreq := &syscall.IPv6Mreq{}
197     copy(mreq.Multiaddr[:], ip)
198     if ifi != nil {
199         mreq.Interface = uint32(ifi.Index)
200     }
201     if err := fd.incref(false); err != nil {
202         return err
203     }
204     defer fd.decref()
205     return os.NewSyscallError("setsockopt", syscall.Sets
206 }
207
208 func leaveIPv6Group(fd *netFD, ifi *Interface, ip IP) error
209     mreq := &syscall.IPv6Mreq{}
210     copy(mreq.Multiaddr[:], ip)
211     if ifi != nil {
212         mreq.Interface = uint32(ifi.Index)
213     }
214     if err := fd.incref(false); err != nil {
215         return err
216     }
217     defer fd.decref()
218     return os.NewSyscallError("setsockopt", syscall.Sets
219 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/sockoptip_linux.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // IP-level socket options for Linux
6
7 package net
8
9 import (
10     "os"
11     "syscall"
12 )
13
14 func ipv4MulticastInterface(fd *netFD) (*Interface, error) {
15     if err := fd.incref(false); err != nil {
16         return nil, err
17     }
18     defer fd.decref()
19     mreq, err := syscall.GetsockoptIPMreqn(fd.sysfd, sys
20     if err != nil {
21         return nil, os.NewSyscallError("getsockopt",
22     }
23     if int(mreq.Ifindex) == 0 {
24         return nil, nil
25     }
26     return InterfaceByIndex(int(mreq.Ifindex))
27 }
28
29 func setIPv4MulticastInterface(fd *netFD, ifi *Interface) er
30     var v int32
31     if ifi != nil {
32         v = int32(ifi.Index)
33     }
34     mreq := &syscall.IPmreqn{Ifindex: v}
35     if err := fd.incref(false); err != nil {
36         return err
37     }
38     defer fd.decref()
39     err := syscall.SetsockoptIPMreqn(fd.sysfd, syscall.I
40     if err != nil {
41         return os.NewSyscallError("setsockopt", err)
```

```

42     }
43     return nil
44 }
45
46 func ipv4MulticastTTL(fd *netFD) (int, error) {
47     if err := fd.incref(false); err != nil {
48         return 0, err
49     }
50     defer fd.decref()
51     v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IPPRO
52     if err != nil {
53         return -1, os.NewSyscallError("getsockopt",
54     }
55     return v, nil
56 }
57
58 func setIPv4MulticastTTL(fd *netFD, v int) error {
59     if err := fd.incref(false); err != nil {
60         return err
61     }
62     defer fd.decref()
63     err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPRO
64     if err != nil {
65         return os.NewSyscallError("setsockopt", err)
66     }
67     return nil
68 }
69
70 func ipv4MulticastLoopback(fd *netFD) (bool, error) {
71     if err := fd.incref(false); err != nil {
72         return false, err
73     }
74     defer fd.decref()
75     v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IP
76     if err != nil {
77         return false, os.NewSyscallError("getsockopt
78     }
79     return v == 1, nil
80 }
81
82 func setIPv4MulticastLoopback(fd *netFD, v bool) error {
83     if err := fd.incref(false); err != nil {
84         return err
85     }
86     defer fd.decref()
87     err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPRO
88     if err != nil {
89         return os.NewSyscallError("setsockopt", err)
90     }
91     return nil

```

```

92 }
93
94 func ipv4ReceiveInterface(fd *netFD) (bool, error) {
95     if err := fd.incref(false); err != nil {
96         return false, err
97     }
98     defer fd.decref()
99     v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IF
100     if err != nil {
101         return false, os.NewSyscallError("getsockopt
102     }
103     return v == 1, nil
104 }
105
106 func setIPv4ReceiveInterface(fd *netFD, v bool) error {
107     if err := fd.incref(false); err != nil {
108         return err
109     }
110     defer fd.decref()
111     err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPRO
112     if err != nil {
113         return os.NewSyscallError("setsockopt", err)
114     }
115     return nil
116 }
117
118 func ipv6TrafficClass(fd *netFD) (int, error) {
119     if err := fd.incref(false); err != nil {
120         return 0, err
121     }
122     defer fd.decref()
123     v, err := syscall.GetsockoptInt(fd.sysfd, syscall.IF
124     if err != nil {
125         return 0, os.NewSyscallError("getsockopt", e
126     }
127     return v, nil
128 }
129
130 func setIPv6TrafficClass(fd *netFD, v int) error {
131     if err := fd.incref(false); err != nil {
132         return err
133     }
134     defer fd.decref()
135     err := syscall.SetsockoptInt(fd.sysfd, syscall.IPPRO
136     if err != nil {
137         return os.NewSyscallError("setsockopt", err)
138     }
139     return nil
140 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/tcpsock.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // TCP sockets
6
7 package net
8
9 // TCPAddr represents the address of a TCP end point.
10 type TCPAddr struct {
11     IP    IP
12     Port int
13 }
14
15 // Network returns the address's network name, "tcp".
16 func (a *TCPAddr) Network() string { return "tcp" }
17
18 func (a *TCPAddr) String() string {
19     if a == nil {
20         return "<nil>"
21     }
22     return JoinHostPort(a.IP.String(), itoa(a.Port))
23 }
24
25 // ResolveTCPAddr parses addr as a TCP address of the form
26 // host:port and resolves domain names or port names to
27 // numeric addresses on the network net, which must be "tcp"
28 // "tcp4" or "tcp6". A literal IPv6 host address must be
29 // enclosed in square brackets, as in "[::]:80".
30 func ResolveTCPAddr(net, addr string) (*TCPAddr, error) {
31     ip, port, err := hostPortToIP(net, addr)
32     if err != nil {
33         return nil, err
34     }
35     return &TCPAddr{ip, port}, nil
36 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/tcpsock_posix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 // TCP sockets
8
9 package net
10
11 import (
12     "io"
13     "os"
14     "syscall"
15     "time"
16 )
17
18 // BUG(rsc): On OpenBSD, listening on the "tcp" network does
19 // both IPv4 and IPv6 connections. This is due to the fact t
20 // will not be routed to an IPv6 socket - two separate socke
21 // if both AFs are to be supported. See inet6(4) on OpenBSD
22
23 func sockaddrToTCP(sa syscall.Sockaddr) Addr {
24     switch sa := sa.(type) {
25     case *syscall.SockaddrInet4:
26         return &TCPAddr{sa.Addr[0:], sa.Port}
27     case *syscall.SockaddrInet6:
28         return &TCPAddr{sa.Addr[0:], sa.Port}
29     default:
30         if sa != nil {
31             // Diagnose when we will turn a non-
32             panic("unexpected type in sockaddrTo
33         }
34     }
35     return nil
36 }
37
38 func (a *TCPAddr) family() int {
39     if a == nil || len(a.IP) <= IPv4len {
40         return syscall.AF_INET
41     }
42 }
```

```

42         if a.IP.To4() != nil {
43             return syscall.AF_INET
44         }
45         return syscall.AF_INET6
46     }
47
48     func (a *TCPAddr) isWildcard() bool {
49         if a == nil || a.IP == nil {
50             return true
51         }
52         return a.IP.IsUnspecified()
53     }
54
55     func (a *TCPAddr) sockaddr(family int) (syscall.Sockaddr, error) {
56         return ipToSockaddr(family, a.IP, a.Port)
57     }
58
59     func (a *TCPAddr) toAddr() sockaddr {
60         if a == nil { // nil *TCPAddr
61             return nil // nil interface
62         }
63         return a
64     }
65
66     // TCPConn is an implementation of the Conn interface
67     // for TCP network connections.
68     type TCPConn struct {
69         fd *netFD
70     }
71
72     func newTCPConn(fd *netFD) *TCPConn {
73         c := &TCPConn{fd}
74         c.SetNoDelay(true)
75         return c
76     }
77
78     func (c *TCPConn) ok() bool { return c != nil && c.fd != nil }
79
80     // Implementation of the Conn interface - see Conn for docu
81
82     // Read implements the Conn Read method.
83     func (c *TCPConn) Read(b []byte) (n int, err error) {
84         if !c.ok() {
85             return 0, syscall.EINVAL
86         }
87         return c.fd.Read(b)
88     }
89
90     // ReadFrom implements the io.ReaderFrom ReadFrom method.
91     func (c *TCPConn) ReadFrom(r io.Reader) (int64, error) {

```

```

92         if n, err, handled := sendFile(c.fd, r); handled {
93             return n, err
94         }
95         return genericReadFrom(c, r)
96     }
97
98     // Write implements the Conn Write method.
99     func (c *TCPConn) Write(b []byte) (n int, err error) {
100         if !c.ok() {
101             return 0, syscall.EINVAL
102         }
103         return c.fd.Write(b)
104     }
105
106     // Close closes the TCP connection.
107     func (c *TCPConn) Close() error {
108         if !c.ok() {
109             return syscall.EINVAL
110         }
111         return c.fd.Close()
112     }
113
114     // CloseRead shuts down the reading side of the TCP connecti
115     // Most callers should just use Close.
116     func (c *TCPConn) CloseRead() error {
117         if !c.ok() {
118             return syscall.EINVAL
119         }
120         return c.fd.CloseRead()
121     }
122
123     // CloseWrite shuts down the writing side of the TCP connect
124     // Most callers should just use Close.
125     func (c *TCPConn) CloseWrite() error {
126         if !c.ok() {
127             return syscall.EINVAL
128         }
129         return c.fd.CloseWrite()
130     }
131
132     // LocalAddr returns the local network address, a *TCPAddr.
133     func (c *TCPConn) LocalAddr() Addr {
134         if !c.ok() {
135             return nil
136         }
137         return c.fd.laddr
138     }
139
140     // RemoteAddr returns the remote network address, a *TCPAddr

```

```

141 func (c *TCPConn) RemoteAddr() Addr {
142     if !c.ok() {
143         return nil
144     }
145     return c.fd.raddr
146 }
147
148 // SetDeadline implements the Conn SetDeadline method.
149 func (c *TCPConn) SetDeadline(t time.Time) error {
150     if !c.ok() {
151         return syscall.EINVAL
152     }
153     return setDeadline(c.fd, t)
154 }
155
156 // SetReadDeadline implements the Conn SetReadDeadline metho
157 func (c *TCPConn) SetReadDeadline(t time.Time) error {
158     if !c.ok() {
159         return syscall.EINVAL
160     }
161     return setReadDeadline(c.fd, t)
162 }
163
164 // SetWriteDeadline implements the Conn SetWriteDeadline met
165 func (c *TCPConn) SetWriteDeadline(t time.Time) error {
166     if !c.ok() {
167         return syscall.EINVAL
168     }
169     return setWriteDeadline(c.fd, t)
170 }
171
172 // SetReadBuffer sets the size of the operating system's
173 // receive buffer associated with the connection.
174 func (c *TCPConn) SetReadBuffer(bytes int) error {
175     if !c.ok() {
176         return syscall.EINVAL
177     }
178     return setReadBuffer(c.fd, bytes)
179 }
180
181 // SetWriteBuffer sets the size of the operating system's
182 // transmit buffer associated with the connection.
183 func (c *TCPConn) SetWriteBuffer(bytes int) error {
184     if !c.ok() {
185         return syscall.EINVAL
186     }
187     return setWriteBuffer(c.fd, bytes)
188 }
189

```

```

190 // SetLinger sets the behavior of Close() on a connection
191 // which still has data waiting to be sent or to be acknowle
192 //
193 // If sec < 0 (the default), Close returns immediately and
194 // the operating system finishes sending the data in the bac
195 //
196 // If sec == 0, Close returns immediately and the operating
197 // discards any unsent or unacknowledged data.
198 //
199 // If sec > 0, Close blocks for at most sec seconds waiting
200 // data to be sent and acknowledged.
201 func (c *TCPConn) SetLinger(sec int) error {
202     if !c.ok() {
203         return syscall.EINVAL
204     }
205     return setLinger(c.fd, sec)
206 }
207
208 // SetKeepAlive sets whether the operating system should sen
209 // keepalive messages on the connection.
210 func (c *TCPConn) SetKeepAlive(keepalive bool) error {
211     if !c.ok() {
212         return syscall.EINVAL
213     }
214     return setKeepAlive(c.fd, keepalive)
215 }
216
217 // SetNoDelay controls whether the operating system should d
218 // packet transmission in hopes of sending fewer packets
219 // (Nagle's algorithm). The default is true (no delay), mea
220 // that data is sent as soon as possible after a Write.
221 func (c *TCPConn) SetNoDelay(noDelay bool) error {
222     if !c.ok() {
223         return syscall.EINVAL
224     }
225     return setNoDelay(c.fd, noDelay)
226 }
227
228 // File returns a copy of the underlying os.File, set to blo
229 // It is the caller's responsibility to close f when finishe
230 // Closing c does not affect f, and closing f does not affec
231 func (c *TCPConn) File() (f *os.File, err error) { return c.
232
233 // DialTCP connects to the remote address raddr on the netwo
234 // which must be "tcp", "tcp4", or "tcp6". If laddr is not
235 // as the local address for the connection.
236 func DialTCP(net string, laddr, raddr *TCPAddr) (*TCPConn, e
237     if raddr == nil {
238         return nil, &OpError{"dial", net, nil, errMi
239     }

```

```

240
241     fd, err := internetSocket(net, laddr.toAddr(), raddr
242
243     // TCP has a rarely used mechanism called a 'simulta
244     // which Dial("tcp", addr1, addr2) run on the machin
245     // connect to a simultaneous Dial("tcp", addr2, addr
246     // at addr2, without either machine executing Listen
247     // it means we want the kernel to pick an appropriat
248     // address. Some Linux kernels cycle blindly throug
249     // local ports, regardless of destination port. If
250     // pick local port 50001 as the source for a Dial("t
251     // then the Dial will succeed, having simultaneously
252     // This can only happen when we are letting the kern
253     // and when there is no listener for the destination
254     // It's hard to argue this is anything other than a
255     // see this happen, rather than expose the buggy eff
256     // close the fd and try again. If it happens twice
257     // use the result. See also:
258     //     http://golang.org/issue/2690
259     //     http://stackoverflow.com/questions/4949858/
260     for i := 0; i < 2 && err == nil && laddr == nil && s
261         fd.Close()
262         fd, err = internetSocket(net, laddr.toAddr())
263     }
264
265     if err != nil {
266         return nil, err
267     }
268     return newTCPConn(fd), nil
269 }
270
271 func selfConnect(fd *netFD) bool {
272     // The socket constructor can return an fd with raddr
273     // unknown conditions. The errors in the calls there
274     // are discarded, but we can't catch the problem the
275     // calls are sometimes legally erroneous with a "soc
276     // Since this code (selfConnect) is already trying t
277     // a problem, we make sure if this happens we recogn
278     // ask the DialTCP routine to try again.
279     // TODO: try to understand what's really going on.
280     if fd.laddr == nil || fd.raddr == nil {
281         return true
282     }
283     l := fd.laddr.(*TCPAddr)
284     r := fd.raddr.(*TCPAddr)
285     return l.Port == r.Port && l.IP.Equal(r.IP)
286 }
287
288 // TCPListener is a TCP network listener.

```

```

289 // Clients should typically use variables of type Listener
290 // instead of assuming TCP.
291 type TCPListener struct {
292     fd *netFD
293 }
294
295 // ListenTCP announces on the TCP address laddr and returns
296 // Net must be "tcp", "tcp4", or "tcp6".
297 // If laddr has a port of 0, it means to listen on some avai
298 // The caller can use l.Addr() to retrieve the chosen address
299 func ListenTCP(net string, laddr *TCPAddr) (*TCPListener, error) {
300     fd, err := internetSocket(net, laddr.toAddr(), nil,
301     if err != nil {
302         return nil, err
303     }
304     err = syscall.Listen(fd.sysfd, listenerBacklog)
305     if err != nil {
306         closesocket(fd.sysfd)
307         return nil, &OpError{"listen", net, laddr, err}
308     }
309     l := new(TCPListener)
310     l.fd = fd
311     return l, nil
312 }
313
314 // AcceptTCP accepts the next incoming call and returns the
315 // and the remote address.
316 func (l *TCPListener) AcceptTCP() (c *TCPConn, error) {
317     if l == nil || l.fd == nil || l.fd.sysfd < 0 {
318         return nil, syscall.EINVAL
319     }
320     fd, err := l.fd.accept(sockaddrToTCP)
321     if err != nil {
322         return nil, err
323     }
324     return newTCPConn(fd), nil
325 }
326
327 // Accept implements the Accept method in the Listener interface
328 // it waits for the next call and returns a generic Conn.
329 func (l *TCPListener) Accept() (c Conn, error) {
330     c1, err := l.AcceptTCP()
331     if err != nil {
332         return nil, err
333     }
334     return c1, nil
335 }
336
337 // Close stops listening on the TCP address.

```

```

338 // Already Accepted connections are not closed.
339 func (l *TCPListener) Close() error {
340     if l == nil || l.fd == nil {
341         return syscall.EINVAL
342     }
343     return l.fd.Close()
344 }
345
346 // Addr returns the listener's network address, a *TCPAddr.
347 func (l *TCPListener) Addr() Addr { return l.fd.laddr }
348
349 // SetDeadline sets the deadline associated with the listener.
350 // A zero time value disables the deadline.
351 func (l *TCPListener) SetDeadline(t time.Time) error {
352     if l == nil || l.fd == nil {
353         return syscall.EINVAL
354     }
355     return setDeadline(l.fd, t)
356 }
357
358 // File returns a copy of the underlying os.File, set to blocking.
359 // It is the caller's responsibility to close f when finished.
360 // Closing l does not affect f, and closing f does not affect l.
361 func (l *TCPListener) File() (f *os.File, err error) { return l.fd.File() }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/udpsock.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // UDP sockets
6
7 package net
8
9 // UDPAddr represents the address of a UDP end point.
10 type UDPAddr struct {
11     IP    IP
12     Port int
13 }
14
15 // Network returns the address's network name, "udp".
16 func (a *UDPAddr) Network() string { return "udp" }
17
18 func (a *UDPAddr) String() string {
19     if a == nil {
20         return "<nil>"
21     }
22     return JoinHostPort(a.IP.String(), itoa(a.Port))
23 }
24
25 // ResolveUDPAddr parses addr as a UDP address of the form
26 // host:port and resolves domain names or port names to
27 // numeric addresses on the network net, which must be "udp"
28 // "udp4" or "udp6". A literal IPv6 host address must be
29 // enclosed in square brackets, as in "[::]:80".
30 func ResolveUDPAddr(net, addr string) (*UDPAddr, error) {
31     ip, port, err := hostPortToIP(net, addr)
32     if err != nil {
33         return nil, err
34     }
35     return &UDPAddr{ip, port}, nil
36 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/udpsock_posix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 // UDP sockets
8
9 package net
10
11 import (
12     "errors"
13     "os"
14     "syscall"
15     "time"
16 )
17
18 var ErrWriteToConnected = errors.New("use of WriteTo with pr
19
20 func sockaddrToUDP(sa syscall.Sockaddr) Addr {
21     switch sa := sa.(type) {
22     case *syscall.SockaddrInet4:
23         return &UDPAddr{sa.Addr[0:], sa.Port}
24     case *syscall.SockaddrInet6:
25         return &UDPAddr{sa.Addr[0:], sa.Port}
26     }
27     return nil
28 }
29
30 func (a *UDPAddr) family() int {
31     if a == nil || len(a.IP) <= IPv4len {
32         return syscall.AF_INET
33     }
34     if a.IP.To4() != nil {
35         return syscall.AF_INET
36     }
37     return syscall.AF_INET6
38 }
39
40 func (a *UDPAddr) isWildcard() bool {
41     if a == nil || a.IP == nil {
```

```

42         return true
43     }
44     return a.IP.IsUnspecified()
45 }
46
47 func (a *UDPAddr) sockaddr(family int) (syscall.Sockaddr, error) {
48     return ipToSockaddr(family, a.IP, a.Port)
49 }
50
51 func (a *UDPAddr) toAddr() syscall.Sockaddr {
52     if a == nil { // nil *UDPAddr
53         return nil // nil interface
54     }
55     return a
56 }
57
58 // UDPConn is the implementation of the Conn and PacketConn
59 // interfaces for UDP network connections.
60 type UDPConn struct {
61     fd *netFD
62 }
63
64 func newUDPConn(fd *netFD) *UDPConn { return &UDPConn{fd} }
65
66 func (c *UDPConn) ok() bool { return c != nil && c.fd != nil }
67
68 // Implementation of the Conn interface - see Conn for docu
69
70 // Read implements the Conn Read method.
71 func (c *UDPConn) Read(b []byte) (int, error) {
72     if !c.ok() {
73         return 0, syscall.EINVAL
74     }
75     return c.fd.Read(b)
76 }
77
78 // Write implements the Conn Write method.
79 func (c *UDPConn) Write(b []byte) (int, error) {
80     if !c.ok() {
81         return 0, syscall.EINVAL
82     }
83     return c.fd.Write(b)
84 }
85
86 // Close closes the UDP connection.
87 func (c *UDPConn) Close() error {
88     if !c.ok() {
89         return syscall.EINVAL
90     }
91     return c.fd.Close()

```

```

92 }
93
94 // LocalAddr returns the local network address.
95 func (c *UDPConn) LocalAddr() Addr {
96     if !c.ok() {
97         return nil
98     }
99     return c.fd.laddr
100 }
101
102 // RemoteAddr returns the remote network address, a *UDPAddr
103 func (c *UDPConn) RemoteAddr() Addr {
104     if !c.ok() {
105         return nil
106     }
107     return c.fd.raddr
108 }
109
110 // SetDeadline implements the Conn SetDeadline method.
111 func (c *UDPConn) SetDeadline(t time.Time) error {
112     if !c.ok() {
113         return syscall.EINVAL
114     }
115     return setDeadline(c.fd, t)
116 }
117
118 // SetReadDeadline implements the Conn SetReadDeadline metho
119 func (c *UDPConn) SetReadDeadline(t time.Time) error {
120     if !c.ok() {
121         return syscall.EINVAL
122     }
123     return setReadDeadline(c.fd, t)
124 }
125
126 // SetWriteDeadline implements the Conn SetWriteDeadline met
127 func (c *UDPConn) SetWriteDeadline(t time.Time) error {
128     if !c.ok() {
129         return syscall.EINVAL
130     }
131     return setWriteDeadline(c.fd, t)
132 }
133
134 // SetReadBuffer sets the size of the operating system's
135 // receive buffer associated with the connection.
136 func (c *UDPConn) SetReadBuffer(bytes int) error {
137     if !c.ok() {
138         return syscall.EINVAL
139     }
140     return setReadBuffer(c.fd, bytes)

```

```

141 }
142
143 // SetWriteBuffer sets the size of the operating system's
144 // transmit buffer associated with the connection.
145 func (c *UDPConn) SetWriteBuffer(bytes int) error {
146     if !c.ok() {
147         return syscall.EINVAL
148     }
149     return setWriteBuffer(c.fd, bytes)
150 }
151
152 // UDP-specific methods.
153
154 // ReadFromUDP reads a UDP packet from c, copying the payload
155 // It returns the number of bytes copied into b and the return
156 // that was on the packet.
157 //
158 // ReadFromUDP can be made to time out and return an error w
159 // after a fixed time limit; see SetDeadline and SetReadDead
160 func (c *UDPConn) ReadFromUDP(b []byte) (n int, addr *UDPAddr
161     if !c.ok() {
162         return 0, nil, syscall.EINVAL
163     }
164     n, sa, err := c.fd.ReadFrom(b)
165     switch sa := sa.(type) {
166     case *syscall.SockaddrInet4:
167         addr = &UDPAddr{sa.Addr[0:], sa.Port}
168     case *syscall.SockaddrInet6:
169         addr = &UDPAddr{sa.Addr[0:], sa.Port}
170     }
171     return
172 }
173
174 // ReadFrom implements the PacketConn ReadFrom method.
175 func (c *UDPConn) ReadFrom(b []byte) (int, Addr, error) {
176     if !c.ok() {
177         return 0, nil, syscall.EINVAL
178     }
179     n, uaddr, err := c.ReadFromUDP(b)
180     return n, uaddr.toAddr(), err
181 }
182
183 // WriteToUDP writes a UDP packet to addr via c, copying the
184 //
185 // WriteToUDP can be made to time out and return
186 // an error with Timeout() == true after a fixed time limit;
187 // see SetDeadline and SetWriteDeadline.
188 // On packet-oriented connections, write timeouts are rare.
189 func (c *UDPConn) WriteToUDP(b []byte, addr *UDPAddr) (int,

```

```

190         if !c.ok() {
191             return 0, syscall.EINVAL
192         }
193         if c.fd.isConnected {
194             return 0, &OpError{"write", c.fd.net, addr,
195             }
196         sa, err := addr.sockaddr(c.fd.family)
197         if err != nil {
198             return 0, &OpError{"write", c.fd.net, addr,
199             }
200         return c.fd.WriteTo(b, sa)
201     }
202
203     // WriteTo implements the PacketConn WriteTo method.
204     func (c *UDPConn) WriteTo(b []byte, addr Addr) (int, error)
205         if !c.ok() {
206             return 0, syscall.EINVAL
207         }
208         a, ok := addr.(*UDPAddr)
209         if !ok {
210             return 0, &OpError{"write", c.fd.net, addr,
211             }
212         return c.WriteToUDP(b, a)
213     }
214
215     // File returns a copy of the underlying os.File, set to blo
216     // It is the caller's responsibility to close f when finishe
217     // Closing c does not affect f, and closing f does not affec
218     func (c *UDPConn) File() (f *os.File, err error) { return c.
219
220     // DialUDP connects to the remote address raddr on the netwo
221     // which must be "udp", "udp4", or "udp6". If laddr is not
222     // as the local address for the connection.
223     func DialUDP(net string, laddr, raddr *UDPAddr) (*UDPConn, e
224         switch net {
225         case "udp", "udp4", "udp6":
226         default:
227             return nil, UnknownNetworkError(net)
228         }
229         if raddr == nil {
230             return nil, &OpError{"dial", net, nil, errMi
231         }
232         fd, err := internetSocket(net, laddr.toAddr(), raddr
233         if err != nil {
234             return nil, err
235         }
236         return newUDPConn(fd), nil
237     }
238
239     // ListenUDP listens for incoming UDP packets addressed to t

```

```

240 // local address laddr. The returned connection c's ReadFro
241 // and WriteTo methods can be used to receive and send UDP
242 // packets with per-packet addressing.
243 func ListenUDP(net string, laddr *UDPAddr) (*UDPCConn, error)
244     switch net {
245     case "udp", "udp4", "udp6":
246     default:
247         return nil, UnknownNetworkError(net)
248     }
249     if laddr == nil {
250         return nil, &OpError{"listen", net, nil, err
251     }
252     fd, err := internetSocket(net, laddr.toAddr(), nil,
253     if err != nil {
254         return nil, err
255     }
256     return newUDPCConn(fd), nil
257 }
258
259 // ListenMulticastUDP listens for incoming multicast UDP pac
260 // addressed to the group address gaddr on ifi, which specif
261 // the interface to join. ListenMulticastUDP uses default
262 // multicast interface if ifi is nil.
263 func ListenMulticastUDP(net string, ifi *Interface, gaddr *U
264     switch net {
265     case "udp", "udp4", "udp6":
266     default:
267         return nil, UnknownNetworkError(net)
268     }
269     if gaddr == nil || gaddr.IP == nil {
270         return nil, &OpError{"listenmulticast", net,
271     }
272     fd, err := internetSocket(net, gaddr.toAddr(), nil,
273     if err != nil {
274         return nil, err
275     }
276     c := newUDPCConn(fd)
277     ip4 := gaddr.IP.To4()
278     if ip4 != nil {
279         err := listenIPv4MulticastUDP(c, ifi, ip4)
280         if err != nil {
281             c.Close()
282             return nil, err
283         }
284     } else {
285         err := listenIPv6MulticastUDP(c, ifi, gaddr.
286         if err != nil {
287             c.Close()
288             return nil, err

```

```

289         }
290     }
291     return c, nil
292 }
293
294 func listenIPv4MulticastUDP(c *UDPCConn, ifi *Interface, ip I
295     if ifi != nil {
296         err := setIPv4MulticastInterface(c.fd, ifi)
297         if err != nil {
298             return err
299         }
300     }
301     err := setIPv4MulticastLoopback(c.fd, false)
302     if err != nil {
303         return err
304     }
305     err = joinIPv4GroupUDP(c, ifi, ip)
306     if err != nil {
307         return err
308     }
309     return nil
310 }
311
312 func listenIPv6MulticastUDP(c *UDPCConn, ifi *Interface, ip I
313     if ifi != nil {
314         err := setIPv6MulticastInterface(c.fd, ifi)
315         if err != nil {
316             return err
317         }
318     }
319     err := setIPv6MulticastLoopback(c.fd, false)
320     if err != nil {
321         return err
322     }
323     err = joinIPv6GroupUDP(c, ifi, ip)
324     if err != nil {
325         return err
326     }
327     return nil
328 }
329
330 func joinIPv4GroupUDP(c *UDPCConn, ifi *Interface, ip IP) err
331     err := joinIPv4Group(c.fd, ifi, ip)
332     if err != nil {
333         return &OpError{"joinipv4group", c.fd.net, &
334     }
335     return nil
336 }
337

```

```
338 func leaveIPv4GroupUDP(c *UDPConn, ifi *Interface, ip IP) er
339     err := leaveIPv4Group(c.fd, ifi, ip)
340     if err != nil {
341         return &OpError{"leaveipv4group", c.fd.net,
342     }
343     return nil
344 }
345
346 func joinIPv6GroupUDP(c *UDPConn, ifi *Interface, ip IP) err
347     err := joinIPv6Group(c.fd, ifi, ip)
348     if err != nil {
349         return &OpError{"joinipv6group", c.fd.net, &
350     }
351     return nil
352 }
353
354 func leaveIPv6GroupUDP(c *UDPConn, ifi *Interface, ip IP) er
355     err := leaveIPv6Group(c.fd, ifi, ip)
356     if err != nil {
357         return &OpError{"leaveipv6group", c.fd.net,
358     }
359     return nil
360 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/unixsock.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Unix domain sockets
6
7 package net
8
9 // UnixAddr represents the address of a Unix domain socket e
10 type UnixAddr struct {
11     Name string
12     Net  string
13 }
14
15 // Network returns the address's network name, "unix" or "un
16 func (a *UnixAddr) Network() string {
17     return a.Net
18 }
19
20 func (a *UnixAddr) String() string {
21     if a == nil {
22         return "<nil>"
23     }
24     return a.Name
25 }
26
27 func (a *UnixAddr) toAddr() Addr {
28     if a == nil { // nil *UnixAddr
29         return nil // nil interface
30     }
31     return a
32 }
33
34 // ResolveUnixAddr parses addr as a Unix domain socket addre
35 // The string net gives the network name, "unix", "unixgram"
36 // "unixpacket".
37 func ResolveUnixAddr(net, addr string) (*UnixAddr, error) {
38     switch net {
39     case "unix":
40     case "unixpacket":
41     case "unixgram":
42     default:
43         return nil, UnknownNetworkError(net)
44     }
```

```
45         return &UnixAddr{addr, net}, nil
46     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/unixsock_posix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 // Unix domain sockets
8
9 package net
10
11 import (
12     "os"
13     "syscall"
14     "time"
15 )
16
17 func unixSocket(net string, laddr, raddr *UnixAddr, mode string) (
18     var sotype int
19     switch net {
20     default:
21         return nil, UnknownNetworkError(net)
22     case "unix":
23         sotype = syscall.SOCK_STREAM
24     case "unixgram":
25         sotype = syscall.SOCK_DGRAM
26     case "unixpacket":
27         sotype = syscall.SOCK_SEQPACKET
28     }
29
30     var la, ra syscall.Sockaddr
31     switch mode {
32     default:
33         panic("unixSocket mode " + mode)
34
35     case "dial":
36         if laddr != nil {
37             la = &syscall.SockaddrUnix{Name: laddr.Name}
38         }
39         if raddr != nil {
40             ra = &syscall.SockaddrUnix{Name: raddr.Name}
41         } else if sotype != syscall.SOCK_DGRAM || la
```

```

42         return nil, &OpError{Op: mode, Net:
43     }
44
45     case "listen":
46         if laddr == nil {
47             return nil, &OpError{mode, net, nil,
48         }
49         la = &syscall.SockaddrUnix{Name: laddr.Name}
50         if raddr != nil {
51             return nil, &OpError{Op: mode, Net:
52         }
53     }
54
55     f := sockaddrToUnix
56     if sotype == syscall.SOCK_DGRAM {
57         f = sockaddrToUnixgram
58     } else if sotype == syscall.SOCK_SEQPACKET {
59         f = sockaddrToUnixpacket
60     }
61
62     fd, err = socket(net, syscall.AF_UNIX, sotype, 0, fa
63     if err != nil {
64         goto Error
65     }
66     return fd, nil
67
68 Error:
69     addr := raddr
70     if mode == "listen" {
71         addr = laddr
72     }
73     return nil, &OpError{Op: mode, Net: net, Addr: addr,
74 }
75
76 func sockaddrToUnix(sa syscall.Sockaddr) Addr {
77     if s, ok := sa.(*syscall.SockaddrUnix); ok {
78         return &UnixAddr{s.Name, "unix"}
79     }
80     return nil
81 }
82
83 func sockaddrToUnixgram(sa syscall.Sockaddr) Addr {
84     if s, ok := sa.(*syscall.SockaddrUnix); ok {
85         return &UnixAddr{s.Name, "unixgram"}
86     }
87     return nil
88 }
89
90 func sockaddrToUnixpacket(sa syscall.Sockaddr) Addr {
91     if s, ok := sa.(*syscall.SockaddrUnix); ok {

```

```

92         return &UnixAddr{s.Name, "unixpacket"}
93     }
94     return nil
95 }
96
97 func sotypeToNet(sotype int) string {
98     switch sotype {
99     case syscall.SOCK_STREAM:
100         return "unix"
101     case syscall.SOCK_SEQPACKET:
102         return "unixpacket"
103     case syscall.SOCK_DGRAM:
104         return "unixgram"
105     default:
106         panic("sotypeToNet unknown socket type")
107     }
108     return ""
109 }
110
111 // UnixConn is an implementation of the Conn interface
112 // for connections to Unix domain sockets.
113 type UnixConn struct {
114     fd *netFD
115 }
116
117 func newUnixConn(fd *netFD) *UnixConn { return &UnixConn{fd} }
118
119 func (c *UnixConn) ok() bool { return c != nil && c.fd != nil }
120
121 // Implementation of the Conn interface - see Conn for documentation
122
123 // Read implements the Conn Read method.
124 func (c *UnixConn) Read(b []byte) (n int, err error) {
125     if !c.ok() {
126         return 0, syscall.EINVAL
127     }
128     return c.fd.Read(b)
129 }
130
131 // Write implements the Conn Write method.
132 func (c *UnixConn) Write(b []byte) (n int, err error) {
133     if !c.ok() {
134         return 0, syscall.EINVAL
135     }
136     return c.fd.Write(b)
137 }
138
139 // Close closes the Unix domain connection.
140 func (c *UnixConn) Close() error {

```

```

141         if !c.ok() {
142             return syscall.EINVAL
143         }
144         return c.fd.Close()
145     }
146
147     // LocalAddr returns the local network address, a *UnixAddr.
148     // Unlike in other protocols, LocalAddr is usually nil for d
149     func (c *UnixConn) LocalAddr() Addr {
150         if !c.ok() {
151             return nil
152         }
153         return c.fd.laddr
154     }
155
156     // RemoteAddr returns the remote network address, a *UnixAddr
157     // Unlike in other protocols, RemoteAddr is usually nil for
158     // accepted by a listener.
159     func (c *UnixConn) RemoteAddr() Addr {
160         if !c.ok() {
161             return nil
162         }
163         return c.fd.raddr
164     }
165
166     // SetDeadline implements the Conn SetDeadline method.
167     func (c *UnixConn) SetDeadline(t time.Time) error {
168         if !c.ok() {
169             return syscall.EINVAL
170         }
171         return setDeadline(c.fd, t)
172     }
173
174     // SetReadDeadline implements the Conn SetReadDeadline metho
175     func (c *UnixConn) SetReadDeadline(t time.Time) error {
176         if !c.ok() {
177             return syscall.EINVAL
178         }
179         return setReadDeadline(c.fd, t)
180     }
181
182     // SetWriteDeadline implements the Conn SetWriteDeadline met
183     func (c *UnixConn) SetWriteDeadline(t time.Time) error {
184         if !c.ok() {
185             return syscall.EINVAL
186         }
187         return setWriteDeadline(c.fd, t)
188     }
189

```

```

190 // SetReadBuffer sets the size of the operating system's
191 // receive buffer associated with the connection.
192 func (c *UnixConn) SetReadBuffer(bytes int) error {
193     if !c.ok() {
194         return syscall.EINVAL
195     }
196     return setReadBuffer(c.fd, bytes)
197 }
198
199 // SetWriteBuffer sets the size of the operating system's
200 // transmit buffer associated with the connection.
201 func (c *UnixConn) SetWriteBuffer(bytes int) error {
202     if !c.ok() {
203         return syscall.EINVAL
204     }
205     return setWriteBuffer(c.fd, bytes)
206 }
207
208 // ReadFromUnix reads a packet from c, copying the payload i
209 // It returns the number of bytes copied into b and the sour
210 // of the packet.
211 //
212 // ReadFromUnix can be made to time out and return
213 // an error with Timeout() == true after a fixed time limit;
214 // see SetDeadline and SetReadDeadline.
215 func (c *UnixConn) ReadFromUnix(b []byte) (n int, addr *Unix
216     if !c.ok() {
217         return 0, nil, syscall.EINVAL
218     }
219     n, sa, err := c.fd.ReadFrom(b)
220     switch sa := sa.(type) {
221     case *syscall.SockaddrUnix:
222         addr = &UnixAddr{sa.Name, sotypeToNet(c.fd.s
223     }
224     return
225 }
226
227 // ReadFrom implements the PacketConn ReadFrom method.
228 func (c *UnixConn) ReadFrom(b []byte) (n int, addr Addr, err
229     if !c.ok() {
230         return 0, nil, syscall.EINVAL
231     }
232     n, uaddr, err := c.ReadFromUnix(b)
233     return n, uaddr.toAddr(), err
234 }
235
236 // WriteToUnix writes a packet to addr via c, copying the pa
237 //
238 // WriteToUnix can be made to time out and return
239 // an error with Timeout() == true after a fixed time limit;

```

```

240 // see SetDeadline and SetWriteDeadline.
241 // On packet-oriented connections, write timeouts are rare.
242 func (c *UnixConn) WriteToUnix(b []byte, addr *UnixAddr) (n
243     if !c.ok() {
244         return 0, syscall.EINVAL
245     }
246     if addr.Net != sotypeToNet(c.fd.sotype) {
247         return 0, syscall.EAFNOSUPPORT
248     }
249     sa := &syscall.SockaddrUnix{Name: addr.Name}
250     return c.fd.WriteTo(b, sa)
251 }
252
253 // WriteTo implements the PacketConn WriteTo method.
254 func (c *UnixConn) WriteTo(b []byte, addr Addr) (n int, err
255     if !c.ok() {
256         return 0, syscall.EINVAL
257     }
258     a, ok := addr.(*UnixAddr)
259     if !ok {
260         return 0, &OpError{"write", c.fd.net, addr,
261     }
262     return c.WriteToUnix(b, a)
263 }
264
265 // ReadMsgUnix reads a packet from c, copying the payload in
266 // and the associated out-of-band data into oob.
267 // It returns the number of bytes copied into b, the number
268 // bytes copied into oob, the flags that were set on the pac
269 // and the source address of the packet.
270 func (c *UnixConn) ReadMsgUnix(b, oob []byte) (n, oobn, flag
271     if !c.ok() {
272         return 0, 0, 0, nil, syscall.EINVAL
273     }
274     n, oobn, flags, sa, err := c.fd.ReadMsg(b, oob)
275     switch sa := sa.(type) {
276     case *syscall.SockaddrUnix:
277         addr = &UnixAddr{sa.Name, sotypeToNet(c.fd.s
278     }
279     return
280 }
281
282 // WriteMsgUnix writes a packet to addr via c, copying the p
283 // and the associated out-of-band data from oob. It returns
284 // of payload and out-of-band bytes written.
285 func (c *UnixConn) WriteMsgUnix(b, oob []byte, addr *UnixAdd
286     if !c.ok() {
287         return 0, 0, syscall.EINVAL
288     }

```

```

289         if addr != nil {
290             if addr.Net != sotypeToNet(c.fd.sotype) {
291                 return 0, 0, syscall.EAFNOSUPPORT
292             }
293             sa := &syscall.SockaddrUnix{Name: addr.Name}
294             return c.fd.WriteMsg(b, oob, sa)
295         }
296         return c.fd.WriteMsg(b, oob, nil)
297     }
298
299     // File returns a copy of the underlying os.File, set to blo
300     // It is the caller's responsibility to close f when finishe
301     // Closing c does not affect f, and closing f does not affec
302     func (c *UnixConn) File() (f *os.File, err error) { return c
303
304     // DialUnix connects to the remote address raddr on the netw
305     // which must be "unix" or "unixgram". If laddr is not nil,
306     // as the local address for the connection.
307     func DialUnix(net string, laddr, raddr *UnixAddr) (*UnixConn
308         fd, err := unixSocket(net, laddr, raddr, "dial")
309         if err != nil {
310             return nil, err
311         }
312         return newUnixConn(fd), nil
313     }
314
315     // UnixListener is a Unix domain socket listener.
316     // Clients should typically use variables of type Listener
317     // instead of assuming Unix domain sockets.
318     type UnixListener struct {
319         fd    *netFD
320         path string
321     }
322
323     // ListenUnix announces on the Unix domain socket laddr and
324     // Net must be "unix" (stream sockets).
325     func ListenUnix(net string, laddr *UnixAddr) (*UnixListener,
326         if net != "unix" && net != "unixgram" && net != "uni
327             return nil, UnknownNetworkError(net)
328         }
329         if laddr != nil {
330             laddr = &UnixAddr{laddr.Name, net} // make o
331         }
332         fd, err := unixSocket(net, laddr, nil, "listen")
333         if err != nil {
334             return nil, err
335         }
336         err = syscall.Listen(fd.sysfd, listenerBacklog)
337         if err != nil {

```

```

338             closesocket(fd.sysfd)
339             return nil, &OpError{Op: "listen", Net: net,
340         }
341         return &UnixListener{fd, laddr.Name}, nil
342     }
343
344     // AcceptUnix accepts the next incoming call and returns the
345     // and the remote address.
346     func (l *UnixListener) AcceptUnix() (*UnixConn, error) {
347         if l == nil || l.fd == nil {
348             return nil, syscall.EINVAL
349         }
350         fd, err := l.fd.accept(sockaddrToUnix)
351         if err != nil {
352             return nil, err
353         }
354         c := newUnixConn(fd)
355         return c, nil
356     }
357
358     // Accept implements the Accept method in the Listener inter
359     // it waits for the next call and returns a generic Conn.
360     func (l *UnixListener) Accept() (c Conn, err error) {
361         c1, err := l.AcceptUnix()
362         if err != nil {
363             return nil, err
364         }
365         return c1, nil
366     }
367
368     // Close stops listening on the Unix address.
369     // Already accepted connections are not closed.
370     func (l *UnixListener) Close() error {
371         if l == nil || l.fd == nil {
372             return syscall.EINVAL
373         }
374
375         // The operating system doesn't clean up
376         // the file that announcing created, so
377         // we have to clean it up ourselves.
378         // There's a race here--we can't know for
379         // sure whether someone else has come along
380         // and replaced our socket name already--
381         // but this sequence (remove then close)
382         // is at least compatible with the auto-remove
383         // sequence in ListenUnix. It's only non-Go
384         // programs that can mess us up.
385         if l.path[0] != '@' {
386             syscall.Unlink(l.path)
387         }

```

```

388         err := l.fd.Close()
389         l.fd = nil
390         return err
391     }
392
393     // Addr returns the listener's network address.
394     func (l *UnixListener) Addr() Addr { return l.fd.laddr }
395
396     // SetDeadline sets the deadline associated with the listener.
397     // A zero time value disables the deadline.
398     func (l *UnixListener) SetDeadline(t time.Time) (err error) {
399         if l == nil || l.fd == nil {
400             return syscall.EINVAL
401         }
402         return setDeadline(l.fd, t)
403     }
404
405     // File returns a copy of the underlying os.File, set to block size.
406     // It is the caller's responsibility to close f when finished.
407     // Closing l does not affect f, and closing f does not affect l.
408     func (l *UnixListener) File() (f *os.File, err error) { return l.fd.File() }
409
410     // ListenUnixgram listens for incoming Unix datagram packets on the
411     // local address laddr. The returned connection c's ReadFrom and
412     // WriteTo methods can be used to receive and send UDP packets with
413     // per-packet addressing. The network net must be "unixgram".
414     func ListenUnixgram(net string, laddr *UnixAddr) (*UDPConn, error) {
415         switch net {
416         case "unixgram":
417             default:
418                 return nil, UnknownNetworkError(net)
419         }
420         if laddr == nil {
421             return nil, &OpError{"listen", net, nil, err}
422         }
423         fd, err := unixSocket(net, laddr, nil, "listen")
424         if err != nil {
425             return nil, err
426         }
427         return newUDPConn(fd), nil
428     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/chunked.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // The wire protocol for HTTP's "chunked" Transfer-Encoding.
6
7 // This code is duplicated in httputil/chunked.go.
8 // Please make any changes in both files.
9
10 package http
11
12 import (
13     "bufio"
14     "bytes"
15     "errors"
16     "io"
17     "strconv"
18 )
19
20 const maxLineLength = 4096 // assumed <= bufio.defaultBufSiz
21
22 var ErrLineTooLong = errors.New("header line too long")
23
24 // newChunkedReader returns a new chunkedReader that transla
25 // out of HTTP "chunked" format before returning it.
26 // The chunkedReader returns io.EOF when the final 0-length
27 //
28 // newChunkedReader is not needed by normal applications. Th
29 // automatically decodes chunking when reading response bodi
30 func newChunkedReader(r io.Reader) io.Reader {
31     br, ok := r.(*bufio.Reader)
32     if !ok {
33         br = bufio.NewReader(r)
34     }
35     return &chunkedReader{r: br}
36 }
37
38 type chunkedReader struct {
39     r    *bufio.Reader
40     n    uint64 // unread bytes in chunk
41     err  error
```

```

42 }
43
44 func (cr *chunkedReader) beginChunk() {
45     // chunk-size CRLF
46     var line string
47     line, cr.err = readLine(cr.r)
48     if cr.err != nil {
49         return
50     }
51     cr.n, cr.err = strconv.ParseUint(line, 16, 64)
52     if cr.err != nil {
53         return
54     }
55     if cr.n == 0 {
56         cr.err = io.EOF
57     }
58 }
59
60 func (cr *chunkedReader) Read(b []uint8) (n int, err error) {
61     if cr.err != nil {
62         return 0, cr.err
63     }
64     if cr.n == 0 {
65         cr.beginChunk()
66         if cr.err != nil {
67             return 0, cr.err
68         }
69     }
70     if uint64(len(b)) > cr.n {
71         b = b[0:cr.n]
72     }
73     n, cr.err = cr.r.Read(b)
74     cr.n -= uint64(n)
75     if cr.n == 0 && cr.err == nil {
76         // end of chunk (CRLF)
77         b := make([]byte, 2)
78         if _, cr.err = io.ReadFull(cr.r, b); cr.err
79             if b[0] != '\r' || b[1] != '\n' {
80                 cr.err = errors.New("malform
81             }
82         }
83     }
84     return n, cr.err
85 }
86
87 // Read a line of bytes (up to \n) from b.
88 // Give up if the line exceeds maxLineLength.
89 // The returned bytes are a pointer into storage in
90 // the bufio, so they are only valid until the next bufio re
91 func readLineBytes(b *bufio.Reader) (p []byte, err error) {

```

```

92         if p, err = b.ReadSlice('\n'); err != nil {
93             // We always know when EOF is coming.
94             // If the caller asked for a line, there sho
95             if err == io.EOF {
96                 err = io.ErrUnexpectedEOF
97             } else if err == bufio.ErrBufferFull {
98                 err = ErrLineTooLong
99             }
100            return nil, err
101        }
102        if len(p) >= maxLineLength {
103            return nil, ErrLineTooLong
104        }
105
106        // Chop off trailing white space.
107        p = bytes.TrimRight(p, " \r\t\n")
108
109        return p, nil
110    }
111
112    // readLineBytes, but convert the bytes into a string.
113    func readLine(b *bufio.Reader) (s string, err error) {
114        p, e := readLineBytes(b)
115        if e != nil {
116            return "", e
117        }
118        return string(p), nil
119    }
120
121    // newChunkedWriter returns a new chunkedWriter that transla
122    // "chunked" format before writing them to w. Closing the re
123    // sends the final 0-length chunk that marks the end of the
124    //
125    // newChunkedWriter is not needed by normal applications. Th
126    // package adds chunking automatically if handlers don't set
127    // Content-Length header. Using newChunkedWriter inside a ha
128    // would result in double chunking or chunking with a Conten
129    // length, both of which are wrong.
130    func newChunkedWriter(w io.Writer) io.WriteCloser {
131        return &chunkedWriter{w}
132    }
133
134    // Writing to chunkedWriter translates to writing in HTTP ch
135    // Encoding wire format to the underlying Wire chunkedWriter
136    type chunkedWriter struct {
137        Wire io.Writer
138    }
139
140    // Write the contents of data as one chunk to Wire.

```

```

141 // NOTE: Note that the corresponding chunk-writing procedure
142 // a bug since it does not check for success of io.WriteString
143 func (cw *chunkedWriter) Write(data []byte) (n int, err error)
144
145     // Don't send 0-length data. It looks like EOF for c
146     if len(data) == 0 {
147         return 0, nil
148     }
149
150     head := strconv.FormatInt(int64(len(data)), 16) + "\
151
152     if _, err = io.WriteString(cw.Wire, head); err != nil
153         return 0, err
154     }
155     if n, err = cw.Wire.Write(data); err != nil {
156         return
157     }
158     if n != len(data) {
159         err = io.ErrShortWrite
160         return
161     }
162     _, err = io.WriteString(cw.Wire, "\r\n")
163
164     return
165 }
166
167 func (cw *chunkedWriter) Close() error {
168     _, err := io.WriteString(cw.Wire, "0\r\n")
169     return err
170 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/client.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // HTTP client. See RFC 2616.
6 //
7 // This is the high-level Client interface.
8 // The low-level implementation is in transport.go.
9
10 package http
11
12 import (
13     "encoding/base64"
14     "errors"
15     "fmt"
16     "io"
17     "net/url"
18     "strings"
19 )
20
21 // A Client is an HTTP client. Its zero value (DefaultClient)
22 // that uses DefaultTransport.
23 //
24 // The Client's Transport typically has internal state (cached
25 // TCP connections), so Clients should be reused instead of
26 // needed. Clients are safe for concurrent use by multiple goroutines.
27 type Client struct {
28     // Transport specifies the mechanism by which individual
29     // HTTP requests are made.
30     // If nil, DefaultTransport is used.
31     Transport RoundTripper
32
33     // CheckRedirect specifies the policy for handling redirects.
34     // If CheckRedirect is not nil, the client calls it
35     // following an HTTP redirect. The arguments req and via
36     // are the upcoming request and the requests made along the way,
37     // oldest first. If CheckRedirect returns an error, the client
38     // returns that error instead of issuing the request.
39     //
40     // If CheckRedirect is nil, the Client uses its default
41     // which is to stop after 10 consecutive requests.
42     CheckRedirect func(req *Request, via []*Request) error
43
44     // Jar specifies the cookie jar.
```

```

45         // If Jar is nil, cookies are not sent in requests a
46         // in responses.
47         Jar CookieJar
48     }
49
50     // DefaultClient is the default Client and is used by Get, H
51     var DefaultClient = &Client{}
52
53     // RoundTripper is an interface representing the ability to
54     // single HTTP transaction, obtaining the Response for a giv
55     //
56     // A RoundTripper must be safe for concurrent use by multipl
57     // goroutines.
58     type RoundTripper interface {
59         // RoundTrip executes a single HTTP transaction, ret
60         // the Response for the request req. RoundTrip shou
61         // attempt to interpret the response. In particular
62         // RoundTrip must return err == nil if it obtained a
63         // regardless of the response's HTTP status code. A
64         // err should be reserved for failure to obtain a re
65         // Similarly, RoundTrip should not attempt to handle
66         // higher-level protocol details such as redirects,
67         // authentication, or cookies.
68         //
69         // RoundTrip should not modify the request, except f
70         // consuming the Body. The request's URL and Header
71         // are guaranteed to be initialized.
72         RoundTrip(*Request) (*Response, error)
73     }
74
75     // Given a string of the form "host", "host:port", or "[ipv6
76     // return true if the string includes a port.
77     func hasPort(s string) bool { return strings.LastIndex(s, ":
78
79     // Used in Send to implement io.ReadCloser by bundling toget
80     // bufio.Reader through which we read the response, and the
81     // network connection.
82     type readClose struct {
83         io.Reader
84         io.Closer
85     }
86
87     // Do sends an HTTP request and returns an HTTP response, fo
88     // policy (e.g. redirects, cookies, auth) as configured on t
89     //
90     // A non-nil response always contains a non-nil resp.Body.
91     //
92     // Callers should close resp.Body when done reading from it.
93     // resp.Body is not closed, the Client's underlying RoundTri
94     // (typically Transport) may not be able to re-use a persist

```

```

95 // connection to the server for a subsequent "keep-alive" re
96 //
97 // Generally Get, Post, or PostForm will be used instead of
98 func (c *Client) Do(req *Request) (resp *Response, err error
99     if req.Method == "GET" || req.Method == "HEAD" {
100         return c.doFollowingRedirects(req)
101     }
102     return send(req, c.Transport)
103 }
104
105 // send issues an HTTP request. Caller should close resp.Body
106 func send(req *Request, t RoundTripper) (resp *Response, err
107     if t == nil {
108         t = DefaultTransport
109         if t == nil {
110             err = errors.New("http: no Client.Transport")
111             return
112         }
113     }
114
115     if req.URL == nil {
116         return nil, errors.New("http: nil Request.URL")
117     }
118
119     if req.RequestURI != "" {
120         return nil, errors.New("http: Request.RequestURI")
121     }
122
123     // Most the callers of send (Get, Post, et al) don't
124     // Headers, leaving it uninitialized. We guarantee
125     // Transport that this has been initialized, though.
126     if req.Header == nil {
127         req.Header = make(Header)
128     }
129
130     if u := req.URL.User; u != nil {
131         req.Header.Set("Authorization", "Basic "+base64.URLEncodeString(u.String()))
132     }
133     return t.RoundTrip(req)
134 }
135
136 // True if the specified HTTP status code is one for which t
137 // automatically redirect.
138 func shouldRedirect(statusCode int) bool {
139     switch statusCode {
140     case StatusMovedPermanently, StatusFound, StatusSeeOther:
141         return true
142     }
143     return false

```

```

144 }
145
146 // Get issues a GET to the specified URL. If the response i
147 // redirect codes, Get follows the redirect, up to a maximum
148 //
149 //     301 (Moved Permanently)
150 //     302 (Found)
151 //     303 (See Other)
152 //     307 (Temporary Redirect)
153 //
154 // Caller should close r.Body when done reading from it.
155 //
156 // Get is a wrapper around DefaultClient.Get.
157 func Get(url string) (r *Response, err error) {
158     return DefaultClient.Get(url)
159 }
160
161 // Get issues a GET to the specified URL. If the response i
162 // following redirect codes, Get follows the redirect after
163 // Client's CheckRedirect function.
164 //
165 //     301 (Moved Permanently)
166 //     302 (Found)
167 //     303 (See Other)
168 //     307 (Temporary Redirect)
169 //
170 // Caller should close r.Body when done reading from it.
171 func (c *Client) Get(url string) (r *Response, err error) {
172     req, err := NewRequest("GET", url, nil)
173     if err != nil {
174         return nil, err
175     }
176     return c.doFollowingRedirects(req)
177 }
178
179 func (c *Client) doFollowingRedirects(ireq *Request) (r *Res
180 // TODO: if/when we add cookie support, the redirect
181 // necessarily supply the same cookies as the origin
182 var base *url.URL
183 redirectChecker := c.CheckRedirect
184 if redirectChecker == nil {
185     redirectChecker = defaultCheckRedirect
186 }
187 var via []*Request
188
189 if ireq.URL == nil {
190     return nil, errors.New("http: nil Request.UR
191 }
192

```

```

193     jar := c.Jar
194     if jar == nil {
195         jar = blackHoleJar{}
196     }
197
198     req := ireq
199     urlStr := "" // next relative or absolute URL to fet
200     for redirect := 0; ; redirect++ {
201         if redirect != 0 {
202             req = new(Request)
203             req.Method = ireq.Method
204             req.Header = make(Header)
205             req.URL, err = base.Parse(urlStr)
206             if err != nil {
207                 break
208             }
209             if len(via) > 0 {
210                 // Add the Referer header.
211                 lastReq := via[len(via)-1]
212                 if lastReq.URL.Scheme != "ht
213                     req.Header.Set("Refe
214             }
215
216             err = redirectChecker(req, v
217             if err != nil {
218                 break
219             }
220         }
221     }
222
223     for _, cookie := range jar.Cookies(req.URL)
224         req.AddCookie(cookie)
225     }
226     urlStr = req.URL.String()
227     if r, err = send(req, c.Transport); err != n
228         break
229     }
230     if c := r.Cookies(); len(c) > 0 {
231         jar.SetCookies(req.URL, c)
232     }
233
234     if shouldRedirect(r.StatusCode) {
235         r.Body.Close()
236         if urlStr = r.Header.Get("Location")
237             err = errors.New(fmt.Sprintf
238                 break
239         }
240         base = req.URL
241         via = append(via, req)
242         continue

```

```

243         }
244         return
245     }
246
247     method := ireq.Method
248     err = &url.Error{
249         Op: method[0:1] + strings.ToLower(method[1:
250         URL: urlStr,
251         Err: err,
252     }
253     return
254 }
255
256 func defaultCheckRedirect(req *Request, via []*Request) error
257     if len(via) >= 10 {
258         return errors.New("stopped after 10 redirect
259     }
260     return nil
261 }
262
263 // Post issues a POST to the specified URL.
264 //
265 // Caller should close r.Body when done reading from it.
266 //
267 // Post is a wrapper around DefaultClient.Post
268 func Post(url string, bodyType string, body io.Reader) (r *R
269     return DefaultClient.Post(url, bodyType, body)
270 }
271
272 // Post issues a POST to the specified URL.
273 //
274 // Caller should close r.Body when done reading from it.
275 func (c *Client) Post(url string, bodyType string, body io.R
276     req, err := NewRequest("POST", url, body)
277     if err != nil {
278         return nil, err
279     }
280     req.Header.Set("Content-Type", bodyType)
281     r, err = send(req, c.Transport)
282     if err == nil && c.Jar != nil {
283         c.Jar.SetCookies(req.URL, r.Cookies())
284     }
285     return r, err
286 }
287
288 // PostForm issues a POST to the specified URL,
289 // with data's keys and values urlencoded as the request bod
290 //
291 // Caller should close r.Body when done reading from it.

```

```

292 //
293 // PostForm is a wrapper around DefaultClient.PostForm
294 func PostForm(url string, data url.Values) (r *Response, err
295         return DefaultClient.PostForm(url, data)
296 }
297
298 // PostForm issues a POST to the specified URL,
299 // with data's keys and values urlencoded as the request bod
300 //
301 // Caller should close r.Body when done reading from it.
302 func (c *Client) PostForm(url string, data url.Values) (r *R
303         return c.Post(url, "application/x-www-form-urlencoded
304 }
305
306 // Head issues a HEAD to the specified URL. If the response
307 // following redirect codes, Head follows the redirect after
308 // Client's CheckRedirect function.
309 //
310 //     301 (Moved Permanently)
311 //     302 (Found)
312 //     303 (See Other)
313 //     307 (Temporary Redirect)
314 //
315 // Head is a wrapper around DefaultClient.Head
316 func Head(url string) (r *Response, err error) {
317         return DefaultClient.Head(url)
318 }
319
320 // Head issues a HEAD to the specified URL. If the response
321 // following redirect codes, Head follows the redirect after
322 // Client's CheckRedirect function.
323 //
324 //     301 (Moved Permanently)
325 //     302 (Found)
326 //     303 (See Other)
327 //     307 (Temporary Redirect)
328 func (c *Client) Head(url string) (r *Response, err error) {
329         req, err := NewRequest("HEAD", url, nil)
330         if err != nil {
331                 return nil, err
332         }
333         return c.doFollowingRedirects(req)
334 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/cookie.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package http
6
7 import (
8     "bytes"
9     "fmt"
10    "strconv"
11    "strings"
12    "time"
13 )
14
15 // This implementation is done according to RFC 6265:
16 //
17 //     http://tools.ietf.org/html/rfc6265
18
19 // A Cookie represents an HTTP cookie as sent in the Set-Coo
20 // HTTP response or the Cookie header of an HTTP request.
21 type Cookie struct {
22     Name      string
23     Value     string
24     Path     string
25     Domain   string
26     Expires  time.Time
27     RawExpires string
28
29     // MaxAge=0 means no 'Max-Age' attribute specified.
30     // MaxAge<0 means delete cookie now, equivalently 'M
31     // MaxAge>0 means Max-Age attribute present and give
32     MaxAge    int
33     Secure   bool
34     HttpOnly bool
35     Raw      string
36     Unparsed []string // Raw text of unparsed attribute-
37 }
38
39 // readSetCookies parses all "Set-Cookie" values from
40 // the header h and returns the successfully parsed Cookies.
41 func readSetCookies(h Header) []*Cookie {
42     cookies := []*Cookie{}
43     for _, line := range h["Set-Cookie"] {
44         parts := strings.Split(strings.TrimSpace(line
```

```

45     if len(parts) == 1 && parts[0] == "" {
46         continue
47     }
48     parts[0] = strings.TrimSpace(parts[0])
49     j := strings.Index(parts[0], "=")
50     if j < 0 {
51         continue
52     }
53     name, value := parts[0][:j], parts[0][j+1:]
54     if !isCookieNameValid(name) {
55         continue
56     }
57     value, success := parseCookieValue(value)
58     if !success {
59         continue
60     }
61     c := &Cookie{
62         Name: name,
63         Value: value,
64         Raw: line,
65     }
66     for i := 1; i < len(parts); i++ {
67         parts[i] = strings.TrimSpace(parts[i])
68         if len(parts[i]) == 0 {
69             continue
70         }
71
72         attr, val := parts[i], ""
73         if j := strings.Index(attr, "="); j
74             attr, val = attr[:j], attr[j:]
75     }
76     lowerAttr := strings.ToLower(attr)
77     parseCookieValueFn := parseCookieVal
78     if lowerAttr == "expires" {
79         parseCookieValueFn = parseCo
80     }
81     val, success = parseCookieValueFn(va
82     if !success {
83         c.Unparsed = append(c.Unpars
84         continue
85     }
86     switch lowerAttr {
87     case "secure":
88         c.Secure = true
89         continue
90     case "httponly":
91         c.HttpOnly = true
92         continue
93     case "domain":
94         c.Domain = val

```

```

95         // TODO: Add domain parsing
96         continue
97     case "max-age":
98         secs, err := strconv.Atoi(va
99         if err != nil || secs != 0 &
100             break
101     }
102     if secs <= 0 {
103         c.MaxAge = -1
104     } else {
105         c.MaxAge = secs
106     }
107     continue
108     case "expires":
109         c.RawExpires = val
110         exptime, err := time.Parse(t
111         if err != nil {
112             exptime, err = time.
113             if err != nil {
114                 c.Expires =
115                 break
116             }
117         }
118         c.Expires = exptime.UTC()
119         continue
120     case "path":
121         c.Path = val
122         // TODO: Add path parsing
123         continue
124     }
125     c.Unparsed = append(c.Unparsed, part
126 }
127     cookies = append(cookies, c)
128 }
129     return cookies
130 }
131
132 // SetCookie adds a Set-Cookie header to the provided Respon
133 func SetCookie(w ResponseWriter, cookie *Cookie) {
134     w.Header().Add("Set-Cookie", cookie.String())
135 }
136
137 // String returns the serialization of the cookie for use in
138 // header (if only Name and Value are set) or a Set-Cookie r
139 // header (if other fields are set).
140 func (c *Cookie) String() string {
141     var b bytes.Buffer
142     fmt.Fprintf(&b, "%s=%s", sanitizeName(c.Name), sanit
143     if len(c.Path) > 0 {

```

```

144         fmt.Fprintf(&b, "; Path=%s", sanitizeValue(c
145     })
146     if len(c.Domain) > 0 {
147         fmt.Fprintf(&b, "; Domain=%s", sanitizeValue
148     })
149     if c.Expires.Unix() > 0 {
150         fmt.Fprintf(&b, "; Expires=%s", c.Expires.UT
151     })
152     if c.MaxAge > 0 {
153         fmt.Fprintf(&b, "; Max-Age=%d", c.MaxAge)
154     } else if c.MaxAge < 0 {
155         fmt.Fprintf(&b, "; Max-Age=0")
156     }
157     if c.HttpOnly {
158         fmt.Fprintf(&b, "; HttpOnly")
159     }
160     if c.Secure {
161         fmt.Fprintf(&b, "; Secure")
162     }
163     return b.String()
164 }
165
166 // readCookies parses all "Cookie" values from the header h
167 // returns the successfully parsed Cookies.
168 //
169 // if filter isn't empty, only cookies of that name are retu
170 func readCookies(h Header, filter string) []*Cookie {
171     cookies := []*Cookie{}
172     lines, ok := h["Cookie"]
173     if !ok {
174         return cookies
175     }
176
177     for _, line := range lines {
178         parts := strings.Split(strings.TrimSpace(line)
179         if len(parts) == 1 && parts[0] == "" {
180             continue
181         }
182         // Per-line attributes
183         parsedPairs := 0
184         for i := 0; i < len(parts); i++ {
185             parts[i] = strings.TrimSpace(parts[i]
186             if len(parts[i]) == 0 {
187                 continue
188             }
189             name, val := parts[i], ""
190             if j := strings.Index(name, "="); j
191                 name, val = name[:j], name[j
192         }

```

```

193         if !isCookieNameValid(name) {
194             continue
195         }
196         if filter != "" && filter != name {
197             continue
198         }
199         val, success := parseCookieValue(val)
200         if !success {
201             continue
202         }
203         cookies = append(cookies, &Cookie{Na
204             parsedPairs++
205         }
206     }
207     return cookies
208 }
209
210 var cookieNameSanitizer = strings.NewReplacer("\n", "-", "\r
211
212 func sanitizeName(n string) string {
213     return cookieNameSanitizer.Replace(n)
214 }
215
216 var cookieValueSanitizer = strings.NewReplacer("\n", " ", "\
217
218 func sanitizeValue(v string) string {
219     return cookieValueSanitizer.Replace(v)
220 }
221
222 func unquoteCookieValue(v string) string {
223     if len(v) > 1 && v[0] == '"' && v[len(v)-1] == '"' {
224         return v[1 : len(v)-1]
225     }
226     return v
227 }
228
229 func isCookieByte(c byte) bool {
230     switch {
231     case c == 0x21, 0x23 <= c && c <= 0x2b, 0x2d <= c &&
232         0x3c <= c && c <= 0x5b, 0x5d <= c && c <= 0x
233         return true
234     }
235     return false
236 }
237
238 func isCookieExpiresByte(c byte) (ok bool) {
239     return isCookieByte(c) || c == ',' || c == ' '
240 }
241
242 func parseCookieValue(raw string) (string, bool) {

```

```

243         return parseCookieValueUsing(raw, isCookieByte)
244     }
245
246     func parseCookieExpiresValue(raw string) (string, bool) {
247         return parseCookieValueUsing(raw, isCookieExpiresByt
248     }
249
250     func parseCookieValueUsing(raw string, validByte func(byte)
251         raw = unquoteCookieValue(raw)
252         for i := 0; i < len(raw); i++ {
253             if !validByte(raw[i]) {
254                 return "", false
255             }
256         }
257         return raw, true
258     }
259
260     func isCookieNameValid(raw string) bool {
261         for _, c := range raw {
262             if !isToken(byte(c)) {
263                 return false
264             }
265         }
266         return true
267     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/doc.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6 Package http provides HTTP client and server implementations
7
8 Get, Head, Post, and PostForm make HTTP requests:
9
10     resp, err := http.Get("http://example.com/")
11     ...
12     resp, err := http.Post("http://example.com/upload",
13     ...
14     resp, err := http.PostForm("http://example.com/form"
15         url.Values{"key": {"Value"}, "id": {"123"}})
16
17 The client must close the response body when finished with it:
18
19     resp, err := http.Get("http://example.com/")
20     if err != nil {
21         // handle error
22     }
23     defer resp.Body.Close()
24     body, err := ioutil.ReadAll(resp.Body)
25     // ...
26
27 For control over HTTP client headers, redirect policy, and other
28 settings, create a Client:
29
30     client := &http.Client{
31         CheckRedirect: redirectPolicyFunc,
32     }
33
34     resp, err := client.Get("http://example.com")
35     // ...
36
37     req, err := http.NewRequest("GET", "http://example.com",
38     // ...
39     req.Header.Add("If-None-Match", `W/"wyzzy"`)
40     resp, err := client.Do(req)
41     // ...
42
43 For control over proxies, TLS configuration, keep-alives,
44 compression, and other settings, create a Transport:
```

```

45
46     tr := &http.Transport{
47         TLSClientConfig: &tls.Config{RootCAs: poc
48         DisableCompression: true,
49     }
50     client := &http.Client{Transport: tr}
51     resp, err := client.Get("https://example.com")
52
53 Clients and Transports are safe for concurrent use by multip
54 goroutines and for efficiency should only be created once an
55
56 ListenAndServe starts an HTTP server with a given address an
57 The handler is usually nil, which means to use DefaultServeM
58 Handle and HandleFunc add handlers to DefaultServeMux:
59
60     http.Handle("/foo", fooHandler)
61
62     http.HandleFunc("/bar", func(w http.ResponseWriter,
63         fmt.Fprintf(w, "Hello, %q", html.EscapeStrin
64     })
65
66     log.Fatal(http.ListenAndServe(":8080", nil))
67
68 More control over the server's behavior is available by crea
69 custom Server:
70
71     s := &http.Server{
72         Addr:         ":8080",
73         Handler:      myHandler,
74         ReadTimeout:  10 * time.Second,
75         WriteTimeout: 10 * time.Second,
76         MaxHeaderBytes: 1 << 20,
77     }
78     log.Fatal(s.ListenAndServe())
79 */
80 package http

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/filetransport.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package http
6
7 import (
8     "fmt"
9     "io"
10 )
11
12 // fileTransport implements RoundTripper for the 'file' prot
13 type fileTransport struct {
14     fh fileHandler
15 }
16
17 // NewFileTransport returns a new RoundTripper, serving the
18 // FileSystem. The returned RoundTripper ignores the URL hos
19 // incoming requests, as well as most other properties of th
20 // request.
21 //
22 // The typical use case for NewFileTransport is to register
23 // protocol with a Transport, as in:
24 //
25 //   t := &http.Transport{}
26 //   t.RegisterProtocol("file", http.NewFileTransport(http.D
27 //   c := &http.Client{Transport: t}
28 //   res, err := c.Get("file:///etc/passwd")
29 //   ...
30 func NewFileTransport(fs FileSystem) RoundTripper {
31     return fileTransport{fileHandler{fs}}
32 }
33
34 func (t fileTransport) RoundTrip(req *Request) (resp *Respon
35     // We start ServeHTTP in a goroutine, which may take
36     // time if the file is large. The newPopulateRespon
37     // call returns a channel which either ServeHTTP or
38     // sends our *Response on, once the *Response itself
39     // populated (even if the body itself is still being
40     // written to the res.Body, a pipe)
41     rw, resc := newPopulateResponseWriter()
```

```

42         go func() {
43             t.fh.ServeHTTP(rw, req)
44             rw.finish()
45         }()
46         return <-resc, nil
47     }
48
49     func newPopulateResponseWriter() (*populateResponse, <-chan
50         pr, pw := io.Pipe()
51         rw := &populateResponse{
52             ch: make(chan *Response),
53             pw: pw,
54             res: &Response{
55                 Proto:         "HTTP/1.0",
56                 ProtoMajor: 1,
57                 Header:         make(Header),
58                 Close:         true,
59                 Body:         pr,
60             },
61         }
62         return rw, rw.ch
63     }
64
65     // populateResponse is a ResponseWriter that populates the *
66     // in res, and writes its body to a pipe connected to the re
67     // body. Once writes begin or finish() is called, the respon
68     // on ch.
69     type populateResponse struct {
70         res          *Response
71         ch            chan *Response
72         wroteHeader  bool
73         hasContent   bool
74         sentResponse bool
75         pw           *io.PipeWriter
76     }
77
78     func (pr *populateResponse) finish() {
79         if !pr.wroteHeader {
80             pr.WriteHeader(500)
81         }
82         if !pr.sentResponse {
83             pr.sendResponse()
84         }
85         pr.pw.Close()
86     }
87
88     func (pr *populateResponse) sendResponse() {
89         if pr.sentResponse {
90             return
91         }

```

```

92         pr.sentResponse = true
93
94         if pr.hasContent {
95             pr.res.ContentLength = -1
96         }
97         pr.ch <- pr.res
98     }
99
100    func (pr *populateResponse) Header() Header {
101        return pr.res.Header
102    }
103
104    func (pr *populateResponse) WriteHeader(code int) {
105        if pr.wroteHeader {
106            return
107        }
108        pr.wroteHeader = true
109
110        pr.res.StatusCode = code
111        pr.res.Status = fmt.Sprintf("%d %s", code, StatusText[code])
112    }
113
114    func (pr *populateResponse) Write(p []byte) (n int, err error) {
115        if !pr.wroteHeader {
116            pr.WriteHeader(StatusOK)
117        }
118        pr.hasContent = true
119        if !pr.sentResponse {
120            pr.sendResponse()
121        }
122        return pr.pw.Write(p)
123    }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/fs.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // HTTP file system request handler
6
7 package http
8
9 import (
10     "errors"
11     "fmt"
12     "io"
13     "mime"
14     "os"
15     "path"
16     "path/filepath"
17     "strconv"
18     "strings"
19     "time"
20 )
21
22 // A Dir implements http.FileSystem using the native file
23 // system restricted to a specific directory tree.
24 //
25 // An empty Dir is treated as ".".
26 type Dir string
27
28 func (d Dir) Open(name string) (File, error) {
29     if filepath.Separator != '/' && strings.IndexRune(name, filepath.Separator) > 0 {
30         return nil, errors.New("http: invalid character " + string(filepath.Separator) + " in file name")
31     }
32     dir := string(d)
33     if dir == "" {
34         dir = "."
35     }
36     f, err := os.Open(filepath.Join(dir, filepath.FromSlash(name)))
37     if err != nil {
38         return nil, err
39     }
40     return f, nil
41 }
42
43 // A FileSystem implements access to a collection of named files.
44 // The elements in a file path are separated by slash ('/'),
```

```

45 // characters, regardless of host operating system conventio
46 type FileSystem interface {
47     Open(name string) (File, error)
48 }
49
50 // A File is returned by a FileSystem's Open method and can
51 // served by the FileServer implementation.
52 type File interface {
53     Close() error
54     Stat() (os.FileInfo, error)
55     Readdir(count int) ([]os.FileInfo, error)
56     Read([]byte) (int, error)
57     Seek(offset int64, whence int) (int64, error)
58 }
59
60 func dirList(w ResponseWriter, f File) {
61     w.Header().Set("Content-Type", "text/html; charset=u
62     fmt.Fprintf(w, "<pre>\n")
63     for {
64         dirs, err := f.Readdir(100)
65         if err != nil || len(dirs) == 0 {
66             break
67         }
68         for _, d := range dirs {
69             name := d.Name()
70             if d.IsDir() {
71                 name += "/"
72             }
73             // TODO htmlescape
74             fmt.Fprintf(w, "<a href=\"%s\">%s</a
75         }
76     }
77     fmt.Fprintf(w, "</pre>\n")
78 }
79
80 // ServeContent replies to the request using the content in
81 // provided ReadSeeker. The main benefit of ServeContent ov
82 // is that it handles Range requests properly, sets the MIME
83 // handles If-Modified-Since requests.
84 //
85 // If the response's Content-Type header is not set, ServeCo
86 // first tries to deduce the type from name's file extension
87 // if that fails, falls back to reading the first block of t
88 // and passing it to DetectContentType.
89 // The name is otherwise unused; in particular it can be emp
90 // never sent in the response.
91 //
92 // If modtime is not the zero time, ServeContent includes it
93 // Last-Modified header in the response. If the request inc
94 // If-Modified-Since header, ServeContent uses modtime to de

```

```

95 // whether the content needs to be sent at all.
96 //
97 // The content's Seek method must work: ServeContent uses
98 // a seek to the end of the content to determine its size.
99 //
100 // Note that *os.File implements the io.ReadSeeker interface
101 func ServeContent(w ResponseWriter, req *Request, name string,
102     size, err := content.Seek(0, os.SEEK_END)
103     if err != nil {
104         Error(w, "seeker can't seek", StatusInternalServerError)
105         return
106     }
107     _, err = content.Seek(0, os.SEEK_SET)
108     if err != nil {
109         Error(w, "seeker can't seek", StatusInternalServerError)
110         return
111     }
112     serveContent(w, req, name, modtime, size, content)
113 }
114
115 // if name is empty, filename is unknown. (used for mime type)
116 // if modtime.IsZero(), modtime is unknown.
117 // content must be seeked to the beginning of the file.
118 func serveContent(w ResponseWriter, r *Request, name string,
119     if checkLastModified(w, r, modtime) {
120         return
121     }
122
123     code := StatusOK
124
125     // If Content-Type isn't set, use the file's extension
126     if w.Header().Get("Content-Type") == "" {
127         ctype := mime.TypeByExtension(filepath.Ext(name))
128         if ctype == "" {
129             // read a chunk to decide between text/html and application/javascript
130             var buf [1024]byte
131             n, _ := io.ReadFull(content, buf[:])
132             b := buf[:n]
133             ctype = DetectContentType(b)
134             _, err := content.Seek(0, os.SEEK_SET)
135             if err != nil {
136                 Error(w, "seeker can't seek")
137                 return
138             }
139         }
140         w.Header().Set("Content-Type", ctype)
141     }
142
143     // handle Content-Range header.

```

```

144 // TODO(adg): handle multiple ranges
145 sendSize := size
146 if size >= 0 {
147     ranges, err := parseRange(r.Header.Get("Range")
148     if err == nil && len(ranges) > 1 {
149         err = errors.New("multiple ranges no
150     }
151     if err != nil {
152         Error(w, err.Error(), StatusRequester
153         return
154     }
155     if len(ranges) == 1 {
156         ra := ranges[0]
157         if _, err := content.Seek(ra.start,
158             Error(w, err.Error(), Status
159             return
160         }
161         sendSize = ra.length
162         code = StatusPartialContent
163         w.Header().Set("Content-Range", fmt.
164     }
165
166     w.Header().Set("Accept-Ranges", "bytes")
167     if w.Header().Get("Content-Encoding") == ""
168         w.Header().Set("Content-Length", str
169     }
170 }
171
172 w.WriteHeader(code)
173
174 if r.Method != "HEAD" {
175     if sendSize == -1 {
176         io.Copy(w, content)
177     } else {
178         io.CopyN(w, content, sendSize)
179     }
180 }
181 }
182
183 // modtime is the modification time of the resource to be se
184 // return value is whether this request is now complete.
185 func checkLastModified(w ResponseWriter, r *Request, modtime
186     if modtime.IsZero() {
187         return false
188     }
189
190 // The Date-Modified header truncates sub-second pre
191 // use mtime < t+1s instead of mtime <= t to check f
192 if t, err := time.Parse(TimeFormat, r.Header.Get("If

```

```

193             w.WriteHeader(StatusNotModified)
194             return true
195         }
196         w.Header().Set("Last-Modified", modtime.UTC().Format
197         return false
198     }
199
200     // name is '/'-separated, not filepath.Separator.
201     func serveFile(w ResponseWriter, r *Request, fs FileSystem,
202         const indexPage = "/index.html"
203
204         // redirect ../index.html to ../
205         // can't use Redirect() because that would make the
206         // which would be a problem running under StripPrefi
207         if strings.HasSuffix(r.URL.Path, indexPage) {
208             localRedirect(w, r, "../")
209             return
210         }
211
212         f, err := fs.Open(name)
213         if err != nil {
214             // TODO expose actual error?
215             NotFound(w, r)
216             return
217         }
218         defer f.Close()
219
220         d, err1 := f.Stat()
221         if err1 != nil {
222             // TODO expose actual error?
223             NotFound(w, r)
224             return
225         }
226
227         if redirect {
228             // redirect to canonical path: / at end of d
229             // r.URL.Path always begins with /
230             url := r.URL.Path
231             if d.IsDir() {
232                 if url[len(url)-1] != '/' {
233                     localRedirect(w, r, path.Base
234                     return
235                 }
236             } else {
237                 if url[len(url)-1] == '/' {
238                     localRedirect(w, r, "../"+pa
239                     return
240                 }
241             }
242         }

```

```

243
244     // use contents of index.html for directory, if pres
245     if d.IsDir() {
246         if checkLastModified(w, r, d.ModTime()) {
247             return
248         }
249         index := name + indexPage
250         ff, err := fs.Open(index)
251         if err == nil {
252             defer ff.Close()
253             dd, err := ff.Stat()
254             if err == nil {
255                 name = index
256                 d = dd
257                 f = ff
258             }
259         }
260     }
261
262     if d.IsDir() {
263         dirList(w, f)
264         return
265     }
266
267     serveContent(w, r, d.Name(), d.ModTime(), d.Size(),
268 }
269
270 // localRedirect gives a Moved Permanently response.
271 // It does not convert relative paths to absolute paths like
272 func localRedirect(w ResponseWriter, r *Request, newPath string) {
273     if q := r.URL.RawQuery; q != "" {
274         newPath += "?" + q
275     }
276     w.Header().Set("Location", newPath)
277     w.WriteHeader(StatusMovedPermanently)
278 }
279
280 // ServeFile replies to the request with the contents of the
281 func ServeFile(w ResponseWriter, r *Request, name string) {
282     dir, file := filepath.Split(name)
283     serveFile(w, r, Dir(dir), file, false)
284 }
285
286 type fileHandler struct {
287     root FileSystem
288 }
289
290 // FileServer returns a handler that serves HTTP requests
291 // with the contents of the file system rooted at root.

```

```

292 //
293 // To use the operating system's file system implementation,
294 // use http.Dir:
295 //
296 //     http.Handle("/", http.FileServer(http.Dir("/tmp")))
297 func FileServer(root FileSystem) Handler {
298     return &fileHandler{root}
299 }
300
301 func (f *fileHandler) ServeHTTP(w ResponseWriter, r *Request
302     upath := r.URL.Path
303     if !strings.HasPrefix(upath, "/") {
304         upath = "/" + upath
305         r.URL.Path = upath
306     }
307     serveFile(w, r, f.root, path.Clean(upath), true)
308 }
309
310 // httpRange specifies the byte range to be sent to the client
311 type httpRange struct {
312     start, length int64
313 }
314
315 // parseRange parses a Range header string as per RFC 2616.
316 func parseRange(s string, size int64) ([]httpRange, error) {
317     if s == "" {
318         return nil, nil // header not present
319     }
320     const b = "bytes="
321     if !strings.HasPrefix(s, b) {
322         return nil, errors.New("invalid range")
323     }
324     var ranges []httpRange
325     for _, ra := range strings.Split(s[len(b):], ",") {
326         i := strings.Index(ra, "-")
327         if i < 0 {
328             return nil, errors.New("invalid range")
329         }
330         start, end := ra[:i], ra[i+1:]
331         var r httpRange
332         if start == "" {
333             // If no start is specified, end specifies
334             // range start relative to the end of the file
335             i, err := strconv.ParseInt(end, 10, 64)
336             if err != nil {
337                 return nil, errors.New("invalid range")
338             }
339             if i > size {
340                 i = size

```

```

341         }
342         r.start = size - i
343         r.length = size - r.start
344     } else {
345         i, err := strconv.ParseInt(start, 10
346         if err != nil || i > size || i < 0 {
347             return nil, errors.New("inva
348         }
349         r.start = i
350         if end == "" {
351             // If no end is specified, r
352             r.length = size - r.start
353         } else {
354             i, err := strconv.ParseInt(e
355             if err != nil || r.start > i
356                 return nil, errors.N
357             }
358             if i >= size {
359                 i = size - 1
360             }
361             r.length = i - r.start + 1
362         }
363     }
364     ranges = append(ranges, r)
365 }
366 return ranges, nil
367 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/header.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package http
6
7 import (
8     "fmt"
9     "io"
10    "net/textproto"
11    "sort"
12    "strings"
13 )
14
15 // A Header represents the key-value pairs in an HTTP header
16 type Header map[string][]string
17
18 // Add adds the key, value pair to the header.
19 // It appends to any existing values associated with key.
20 func (h Header) Add(key, value string) {
21     textproto.MIMEHeader(h).Add(key, value)
22 }
23
24 // Set sets the header entries associated with key to
25 // the single element value. It replaces any existing
26 // values associated with key.
27 func (h Header) Set(key, value string) {
28     textproto.MIMEHeader(h).Set(key, value)
29 }
30
31 // Get gets the first value associated with the given key.
32 // If there are no values associated with the key, Get returns
33 // the empty string. To access multiple values of a key, access the map directly
34 // with CanonicalHeaderKey.
35 func (h Header) Get(key string) string {
36     return textproto.MIMEHeader(h).Get(key)
37 }
38
39 // Del deletes the values associated with key.
40 func (h Header) Del(key string) {
41     textproto.MIMEHeader(h).Del(key)
42 }
43
44 // Write writes a header in wire format.
```

```

45 func (h Header) Write(w io.Writer) error {
46     return h.WriteSubset(w, nil)
47 }
48
49 var headerNewlineToSpace = strings.NewReplacer("\n", " ", "\
50
51 // WriteSubset writes a header in wire format.
52 // If exclude is not nil, keys where exclude[key] == true ar
53 func (h Header) WriteSubset(w io.Writer, exclude map[string]
54     keys := make([]string, 0, len(h))
55     for k := range h {
56         if exclude == nil || !exclude[k] {
57             keys = append(keys, k)
58         }
59     }
60     sort.Strings(keys)
61     for _, k := range keys {
62         for _, v := range h[k] {
63             v = headerNewlineToSpace.Replace(v)
64             v = strings.TrimSpace(v)
65             if _, err := fmt.Fprintf(w, "%s: %s\
66                 return err
67             }
68         }
69     }
70     return nil
71 }
72
73 // CanonicalHeaderKey returns the canonical format of the
74 // header key s. The canonicalization converts the first
75 // letter and any letter following a hyphen to upper case;
76 // the rest are converted to lowercase. For example, the
77 // canonical key for "accept-encoding" is "Accept-Encoding".
78 func CanonicalHeaderKey(s string) string { return textproto.

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/jar.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package http
6
7 import (
8     "net/url"
9 )
10
11 // A CookieJar manages storage and use of cookies in HTTP re
12 //
13 // Implementations of CookieJar must be safe for concurrent
14 // goroutines.
15 type CookieJar interface {
16     // SetCookies handles the receipt of the cookies in
17     // given URL. It may or may not choose to save the
18     // on the jar's policy and implementation.
19     SetCookies(u *url.URL, cookies []*Cookie)
20
21     // Cookies returns the cookies to send in a request
22     // It is up to the implementation to honor the stand
23     // restrictions such as in RFC 6265.
24     Cookies(u *url.URL) []*Cookie
25 }
26
27 type blackHoleJar struct{}
28
29 func (blackHoleJar) SetCookies(u *url.URL, cookies []*Cookie)
30 func (blackHoleJar) Cookies(u *url.URL) []*Cookie
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/lex.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package http
6
7 // This file deals with lexical matters of HTTP
8
9 func isSeparator(c byte) bool {
10     switch c {
11     case '(', ')', '<', '>', '@', ',', ';', ':', '\\', '
12         return true
13     }
14     return false
15 }
16
17 func isCtl(c byte) bool { return (0 <= c && c <= 31) || c ==
18
19 func isChar(c byte) bool { return 0 <= c && c <= 127 }
20
21 func isAnyText(c byte) bool { return !isCtl(c) }
22
23 func isQdText(c byte) bool { return isAnyText(c) && c != '"'
24
25 func isToken(c byte) bool { return isChar(c) && !isCtl(c) &&
26
27 // Valid escaped sequences are not specified in RFC 2616, so
28 // that they coincide with the common sense ones used by GO.
29 // characters should probably not be treated as errors by a
30 // parser, so we replace them with the '?' character.
31 func httpUnquotePair(b byte) byte {
32     // skip the first byte, which should always be '\'
33     switch b {
34     case 'a':
35         return '\a'
36     case 'b':
37         return '\b'
38     case 'f':
39         return '\f'
40     case 'n':
41         return '\n'
42     case 'r':
43         return '\r'
44     case 't':
```

```

45         return '\t'
46     case 'v':
47         return '\v'
48     case '\\':
49         return '\\'
50     case '\':
51         return '\'
52     case '"':
53         return '"'
54     }
55     return '?'
56 }
57
58 // raw must begin with a valid quoted string. Only the first
59 // parsed and is unquoted in result. eaten is the number of
60 // upon failure.
61 func httpUnquote(raw []byte) (eaten int, result string) {
62     buf := make([]byte, len(raw))
63     if raw[0] != '"' {
64         return -1, ""
65     }
66     eaten = 1
67     j := 0 // # of bytes written in buf
68     for i := 1; i < len(raw); i++ {
69         switch b := raw[i]; b {
70             case '"':
71                 eaten++
72                 buf = buf[0:j]
73                 return i + 1, string(buf)
74             case '\\':
75                 if len(raw) < i+2 {
76                     return -1, ""
77                 }
78                 buf[j] = httpUnquotePair(raw[i+1])
79                 eaten += 2
80                 j++
81                 i++
82             default:
83                 if isQdText(b) {
84                     buf[j] = b
85                 } else {
86                     buf[j] = '?'
87                 }
88                 eaten++
89                 j++
90         }
91     }
92     return -1, ""
93 }
94

```

```

95 // This is a best effort parse, so errors are not returned,
96 // the input string might be parsed. result is always non-nil
97 func httpSplitFieldValue(fv string) (eaten int, result []string)
98     result = make([]string, 0, len(fv))
99     raw := []byte(fv)
100     i := 0
101     chunk := ""
102     for i < len(raw) {
103         b := raw[i]
104         switch {
105             case b == '"':
106                 eaten, unq := httpUnquote(raw[i:len(
107                     if eaten < 0 {
108                         return i, result
109                     } else {
110                         i += eaten
111                         chunk += unq
112                     }
113                 case isSeparator(b):
114                     if chunk != "" {
115                         result = result[0 : len(result)
116                         result[len(result)-1] = chunk
117                         chunk = ""
118                     }
119                     i++
120                 case isToken(b):
121                     chunk += string(b)
122                     i++
123                 case b == '\n' || b == '\r':
124                     i++
125                 default:
126                     chunk += "?"
127                     i++
128             }
129         }
130         if chunk != "" {
131             result = result[0 : len(result)+1]
132             result[len(result)-1] = chunk
133             chunk = ""
134         }
135         return i, result
136     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/request.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // HTTP Request reading and parsing.
6
7 package http
8
9 import (
10     "bufio"
11     "bytes"
12     "crypto/tls"
13     "encoding/base64"
14     "errors"
15     "fmt"
16     "io"
17     "io/ioutil"
18     "mime"
19     "mime/multipart"
20     "net/textproto"
21     "net/url"
22     "strings"
23 )
24
25 const (
26     maxValueLength = 4096
27     maxHeaderLines = 1024
28     chunkSize      = 4 << 10 // 4 KB chunks
29     defaultMaxMemory = 32 << 20 // 32 MB
30 )
31
32 // ErrMissingFile is returned by FormFile when the provided
33 // is either not present in the request or not a file field.
34 var ErrMissingFile = errors.New("http: no such file")
35
36 // HTTP request parsing errors.
37 type ProtocolError struct {
38     ErrorString string
39 }
40
41 func (err *ProtocolError) Error() string { return err.ErrorS
42
43 var (
44     ErrHeaderTooLong = &ProtocolError{"header too
```

```

45     ErrShortBody           = &ProtocolError{"entity bod
46     ErrNotSupported        = &ProtocolError{"feature no
47     ErrUnexpectedTrailer   = &ProtocolError{"trailer he
48     ErrMissingContentLength = &ProtocolError{"missing Co
49     ErrNotMultipart        = &ProtocolError{"request Co
50     ErrMissingBoundary     = &ProtocolError{"no multipa
51 )
52
53 type badStringError struct {
54     what string
55     str  string
56 }
57
58 func (e *badStringError) Error() string { return fmt.Sprintf
59
60 // Headers that Request.Write handles itself and should be s
61 var reqWriteExcludeHeader = map[string]bool{
62     "Host":           true, // not in Header map anyw
63     "User-Agent":     true,
64     "Content-Length": true,
65     "Transfer-Encoding": true,
66     "Trailer":        true,
67 }
68
69 // A Request represents an HTTP request received by a server
70 // or to be sent by a client.
71 type Request struct {
72     Method string // GET, POST, PUT, etc.
73     URL     *url.URL
74
75     // The protocol version for incoming requests.
76     // Outgoing requests always use HTTP/1.1.
77     Proto      string // "HTTP/1.0"
78     ProtoMajor int    // 1
79     ProtoMinor int    // 0
80
81     // A header maps request lines to their values.
82     // If the header says
83     //
84     //     accept-encoding: gzip, deflate
85     //     Accept-Language: en-us
86     //     Connection: keep-alive
87     //
88     // then
89     //
90     //     Header = map[string][]string{
91     //         "Accept-Encoding": {"gzip, deflate"},
92     //         "Accept-Language": {"en-us"},
93     //         "Connection": {"keep-alive"},
94     //     }

```

```
95 //
96 // HTTP defines that header names are case-insensiti
97 // The request parser implements this by canonicaliz
98 // name, making the first character and any characte
99 // following a hyphen uppercase and the rest lowerca
100 Header Header
101
102 // The message body.
103 Body io.ReadCloser
104
105 // ContentLength records the length of the associate
106 // The value -1 indicates that the length is unknown
107 // Values >= 0 indicate that the given number of byt
108 // be read from Body.
109 // For outgoing requests, a value of 0 means unknown
110 ContentLength int64
111
112 // TransferEncoding lists the transfer encodings fro
113 // innermost. An empty list denotes the "identity" e
114 // TransferEncoding can usually be ignored; chunked
115 // automatically added and removed as necessary when
116 // receiving requests.
117 TransferEncoding []string
118
119 // Close indicates whether to close the connection a
120 // replying to this request.
121 Close bool
122
123 // The host on which the URL is sought.
124 // Per RFC 2616, this is either the value of the Hos
125 // or the host name given in the URL itself.
126 Host string
127
128 // Form contains the parsed form data, including bot
129 // field's query parameters and the POST or PUT form
130 // This field is only available after ParseForm is c
131 // The HTTP client ignores Form and uses Body instea
132 Form url.Values
133
134 // MultipartForm is the parsed multipart form, inclu
135 // This field is only available after ParseMultipart
136 // The HTTP client ignores MultipartForm and uses Bo
137 MultipartForm *multipart.Form
138
139 // Trailer maps trailer keys to values. Like for He
140 // response has multiple trailer lines with the same
141 // concatenated, delimited by commas.
142 // For server requests, Trailer is only populated af
143 // closed or fully consumed.
```

```

144 // Trailer support is only partially complete.
145 Trailer Header
146
147 // RemoteAddr allows HTTP servers and other software
148 // the network address that sent the request, usuall
149 // logging. This field is not filled in by ReadReque
150 // has no defined format. The HTTP server in this pa
151 // sets RemoteAddr to an "IP:port" address before in
152 // handler.
153 // This field is ignored by the HTTP client.
154 RemoteAddr string
155
156 // RequestURI is the unmodified Request-URI of the
157 // Request-Line (RFC 2616, Section 5.1) as sent by t
158 // to a server. Usually the URL field should be used
159 // It is an error to set this field in an HTTP clien
160 RequestURI string
161
162 // TLS allows HTTP servers and other software to rec
163 // information about the TLS connection on which the
164 // was received. This field is not filled in by Read
165 // The HTTP server in this package sets the field fo
166 // TLS-enabled connections before invoking a handler
167 // otherwise it leaves the field nil.
168 // This field is ignored by the HTTP client.
169 TLS *tls.ConnectionState
170 }
171
172 // ProtoAtLeast returns whether the HTTP protocol used
173 // in the request is at least major.minor.
174 func (r *Request) ProtoAtLeast(major, minor int) bool {
175     return r.ProtoMajor > major ||
176         r.ProtoMajor == major && r.ProtoMinor >= min
177 }
178
179 // UserAgent returns the client's User-Agent, if sent in the
180 func (r *Request) UserAgent() string {
181     return r.Header.Get("User-Agent")
182 }
183
184 // Cookies parses and returns the HTTP cookies sent with the
185 func (r *Request) Cookies() []*Cookie {
186     return readCookies(r.Header, "")
187 }
188
189 var ErrNoCookie = errors.New("http: named cookie not present")
190
191 // Cookie returns the named cookie provided in the request o
192 // ErrNoCookie if not found.

```

```

193 func (r *Request) Cookie(name string) (*Cookie, error) {
194     for _, c := range readCookies(r.Header, name) {
195         return c, nil
196     }
197     return nil, ErrNoCookie
198 }
199
200 // AddCookie adds a cookie to the request. Per RFC 6265 sec
201 // AddCookie does not attach more than one Cookie header fie
202 // means all cookies, if any, are written into the same line
203 // separated by semicolon.
204 func (r *Request) AddCookie(c *Cookie) {
205     s := fmt.Sprintf("%s=%s", sanitizeName(c.Name), sani
206     if c := r.Header.Get("Cookie"); c != "" {
207         r.Header.Set("Cookie", c+"; "+s)
208     } else {
209         r.Header.Set("Cookie", s)
210     }
211 }
212
213 // Referer returns the referring URL, if sent in the request
214 //
215 // Referer is misspelled as in the request itself, a mistake
216 // earliest days of HTTP. This value can also be fetched fr
217 // Header map as Header["Referer"]; the benefit of making it
218 // as a method is that the compiler can diagnose programs th
219 // alternate (correct English) spelling req.Referer() but c
220 // diagnose programs that use Header["Referrer"].
221 func (r *Request) Referer() string {
222     return r.Header.Get("Referer")
223 }
224
225 // multipartByReader is a sentinel value.
226 // Its presence in Request.MultipartForm indicates that pars
227 // body has been handed off to a MultipartReader instead of
228 var multipartByReader = &multipart.Form{
229     Value: make(map[string][]string),
230     File:  make(map[string][]*multipart.FileHeader),
231 }
232
233 // MultipartReader returns a MIME multipart reader if this i
234 // multipart/form-data POST request, else returns nil and an
235 // Use this function instead of ParseMultipartForm to
236 // process the request body as a stream.
237 func (r *Request) MultipartReader() (*multipart.Reader, erro
238     if r.MultipartForm == multipartByReader {
239         return nil, errors.New("http: MultipartReade
240     }
241     if r.MultipartForm != nil {
242         return nil, errors.New("http: multipart hand

```

```

243     }
244     r.MultipartForm = multipartByReader
245     return r.multipartReader()
246 }
247
248 func (r *Request) multipartReader() (*multipart.Reader, error) {
249     v := r.Header.Get("Content-Type")
250     if v == "" {
251         return nil, ErrNotMultipart
252     }
253     d, params, err := mime.ParseMediaType(v)
254     if err != nil || d != "multipart/form-data" {
255         return nil, ErrNotMultipart
256     }
257     boundary, ok := params["boundary"]
258     if !ok {
259         return nil, ErrMissingBoundary
260     }
261     return multipart.NewReader(r.Body, boundary), nil
262 }
263
264 // Return value if nonempty, def otherwise.
265 func valueOrDefault(value, def string) string {
266     if value != "" {
267         return value
268     }
269     return def
270 }
271
272 const defaultUserAgent = "Go http package"
273
274 // Write writes an HTTP/1.1 request -- header and body -- in
275 // This method consults the following fields of the request:
276 //     Host
277 //     URL
278 //     Method (defaults to "GET")
279 //     Header
280 //     ContentLength
281 //     TransferEncoding
282 //     Body
283 //
284 // If Body is present, Content-Length is <= 0 and TransferEn
285 // hasn't been set to "identity", Write adds "Transfer-Encod
286 // chunked" to the header. Body is closed after it is sent.
287 func (r *Request) Write(w io.Writer) error {
288     return r.write(w, false, nil)
289 }
290
291 // WriteProxy is like Write but writes the request in the fo

```

```

292 // expected by an HTTP proxy. In particular, WriteProxy wri
293 // initial Request-URI line of the request with an absolute
294 // section 5.1.2 of RFC 2616, including the scheme and host.
295 // In either case, WriteProxy also writes a Host header, usi
296 // either r.Host or r.URL.Host.
297 func (r *Request) WriteProxy(w io.Writer) error {
298     return r.write(w, true, nil)
299 }
300
301 // extraHeaders may be nil
302 func (req *Request) write(w io.Writer, usingProxy bool, extr
303     host := req.Host
304     if host == "" {
305         if req.URL == nil {
306             return errors.New("http: Request.Wri
307         }
308         host = req.URL.Host
309     }
310
311     ruri := req.URL.RequestURI()
312     if usingProxy && req.URL.Scheme != "" && req.URL.Opa
313         ruri = req.URL.Scheme + "://" + host + ruri
314     } else if req.Method == "CONNECT" && req.URL.Path ==
315         // CONNECT requests normally give just the h
316         ruri = host
317     }
318     // TODO(bradfitz): escape at least newlines in ruri?
319
320     bw := bufio.NewWriter(w)
321     fmt.Fprintf(bw, "%s %s HTTP/1.1\r\n", valueOrDefault
322
323     // Header lines
324     fmt.Fprintf(bw, "Host: %s\r\n", host)
325
326     // Use the defaultUserAgent unless the Header contain
327     // may be blank to not send the header.
328     userAgent := defaultUserAgent
329     if req.Header != nil {
330         if ua := req.Header["User-Agent"]; len(ua) >
331             userAgent = ua[0]
332     }
333
334     if userAgent != "" {
335         fmt.Fprintf(bw, "User-Agent: %s\r\n", userAg
336     }
337
338     // Process Body, ContentLength, Close, Trailer
339     tw, err := newTransferWriter(req)
340     if err != nil {

```

```

341         return err
342     }
343     err = tw.WriteHeader(bw)
344     if err != nil {
345         return err
346     }
347
348     // TODO: split long values? (If so, should share co
349     err = req.Header.WriteSubset(bw, reqWriteExcludeHead
350     if err != nil {
351         return err
352     }
353
354     if extraHeaders != nil {
355         err = extraHeaders.Write(bw)
356         if err != nil {
357             return err
358         }
359     }
360
361     io.WriteString(bw, "\r\n")
362
363     // Write body and trailer
364     err = tw.WriteBody(bw)
365     if err != nil {
366         return err
367     }
368
369     return bw.Flush()
370 }
371
372 // Convert decimal at s[i:len(s)] to integer,
373 // returning value, string position where the digits stopped
374 // and whether there was a valid number (digits, not too big
375 func atoi(s string, i int) (n, i1 int, ok bool) {
376     const Big = 1000000
377     if i >= len(s) || s[i] < '0' || s[i] > '9' {
378         return 0, 0, false
379     }
380     n = 0
381     for ; i < len(s) && '0' <= s[i] && s[i] <= '9'; i++
382         n = n*10 + int(s[i]-'0')
383         if n > Big {
384             return 0, 0, false
385         }
386     }
387     return n, i, true
388 }
389
390 // ParseHTTPVersion parses a HTTP version string.

```

```

391 // "HTTP/1.0" returns (1, 0, true).
392 func ParseHTTPVersion(vers string) (major, minor int, ok bool) {
393     if len(vers) < 5 || vers[0:5] != "HTTP/" {
394         return 0, 0, false
395     }
396     major, i, ok := atoi(vers, 5)
397     if !ok || i >= len(vers) || vers[i] != '.' {
398         return 0, 0, false
399     }
400     minor, i, ok = atoi(vers, i+1)
401     if !ok || i != len(vers) {
402         return 0, 0, false
403     }
404     return major, minor, true
405 }
406
407 // NewRequest returns a new Request given a method, URL, and
408 func NewRequest(method, urlStr string, body io.Reader) (*Request, error) {
409     u, err := url.Parse(urlStr)
410     if err != nil {
411         return nil, err
412     }
413     rc, ok := body.(io.ReadCloser)
414     if !ok && body != nil {
415         rc = ioutil.NopCloser(body)
416     }
417     req := &Request{
418         Method:    method,
419         URL:        u,
420         Proto:     "HTTP/1.1",
421         ProtoMajor: 1,
422         ProtoMinor: 1,
423         Header:    make(Header),
424         Body:      rc,
425         Host:     u.Host,
426     }
427     if body != nil {
428         switch v := body.(type) {
429             case *strings.Reader:
430                 req.ContentLength = int64(v.Len())
431             case *bytes.Buffer:
432                 req.ContentLength = int64(v.Len())
433         }
434     }
435     return req, nil
436 }
437
438
439 // SetBasicAuth sets the request's Authorization header to u

```

```

440 // Basic Authentication with the provided username and passw
441 //
442 // With HTTP Basic Authentication the provided username and
443 // are not encrypted.
444 func (r *Request) SetBasicAuth(username, password string) {
445     s := username + ":" + password
446     r.Header.Set("Authorization", "Basic "+base64.StdEnc
447 }
448
449 // ReadRequest reads and parses a request from b.
450 func ReadRequest(b *bufio.Reader) (req *Request, err error)
451
452     tp := textproto.NewReader(b)
453     req = new(Request)
454
455     // First line: GET /index.html HTTP/1.0
456     var s string
457     if s, err = tp.ReadLine(); err != nil {
458         return nil, err
459     }
460     defer func() {
461         if err == io.EOF {
462             err = io.ErrUnexpectedEOF
463         }
464     }()
465
466     var f []string
467     if f = strings.SplitN(s, " ", 3); len(f) < 3 {
468         return nil, &badStringError{"malformed HTTP
469     }
470     req.Method, req.RequestURI, req.Proto = f[0], f[1],
471     rawurl := req.RequestURI
472     var ok bool
473     if req.ProtoMajor, req.ProtoMinor, ok = ParseHTTPVer
474         return nil, &badStringError{"malformed HTTP
475     }
476
477     // CONNECT requests are used two different ways, and
478     // The standard use is to tunnel HTTPS through an HT
479     // It looks like "CONNECT www.google.com:443 HTTP/1.
480     // just the authority section of a URL. This informa
481     //
482     // The net/rpc package also uses CONNECT, but there
483     // that starts with a slash. It can be parsed with t
484     // and the path will end up in req.URL.Path, where i
485     // RPC to work.
486     justAuthority := req.Method == "CONNECT" && !strings
487     if justAuthority {
488         rawurl = "http://" + rawurl

```

```

489     }
490
491     if req.URL, err = url.ParseRequestURI(rawurl); err != nil {
492         return nil, err
493     }
494
495     if justAuthority {
496         // Strip the bogus "http://" back off.
497         req.URL.Scheme = ""
498     }
499
500     // Subsequent lines: Key: value.
501     mimeHeader, err := tp.ReadMIMEHeader()
502     if err != nil {
503         return nil, err
504     }
505     req.Header = Header(mimeHeader)
506
507     // RFC2616: Must treat
508     //     GET /index.html HTTP/1.1
509     //     Host: www.google.com
510     // and
511     //     GET http://www.google.com/index.html HTTP/1.1
512     //     Host: doesntmatter
513     // the same. In the second case, any Host line is ignored.
514     req.Host = req.URL.Host
515     if req.Host == "" {
516         req.Host = req.Header.Get("Host")
517     }
518     req.Header.Del("Host")
519
520     fixPragmaCacheControl(req.Header)
521
522     // TODO: Parse specific header values:
523     //     Accept
524     //     Accept-Encoding
525     //     Accept-Language
526     //     Authorization
527     //     Cache-Control
528     //     Connection
529     //     Date
530     //     Expect
531     //     From
532     //     If-Match
533     //     If-Modified-Since
534     //     If-None-Match
535     //     If-Range
536     //     If-Unmodified-Since
537     //     Max-Forwards
538     //     Proxy-Authorization

```

```

539         //      Referer [sic]
540         //      TE (transfer-codings)
541         //      Trailer
542         //      Transfer-Encoding
543         //      Upgrade
544         //      User-Agent
545         //      Via
546         //      Warning
547
548         err = readTransfer(req, b)
549         if err != nil {
550             return nil, err
551         }
552
553         return req, nil
554     }
555
556     // MaxBytesReader is similar to io.LimitReader but is intend
557     // limiting the size of incoming request bodies. In contrast
558     // io.LimitReader, MaxBytesReader's result is a ReadCloser,
559     // non-EOF error for a Read beyond the limit, and Closes the
560     // underlying reader when its Close method is called.
561     //
562     // MaxBytesReader prevents clients from accidentally or mali
563     // sending a large request and wasting server resources.
564     func MaxBytesReader(w ResponseWriter, r io.ReadCloser, n int
565         return &maxBytesReader{w: w, r: r, n: n}
566     }
567
568     type maxBytesReader struct {
569         w      ResponseWriter
570         r      io.ReadCloser // underlying reader
571         n      int64         // max bytes remaining
572         stopped bool
573     }
574
575     func (l *maxBytesReader) Read(p []byte) (n int, err error) {
576         if l.n <= 0 {
577             if !l.stopped {
578                 l.stopped = true
579                 if res, ok := l.w.(*response); ok {
580                     res.requestTooLarge()
581                 }
582             }
583             return 0, errors.New("http: request body too
584         }
585         if int64(len(p)) > l.n {
586             p = p[:l.n]
587         }

```

```

588         n, err = l.r.Read(p)
589         l.n -= int64(n)
590         return
591     }
592
593     func (l *maxBytesReader) Close() error {
594         return l.r.Close()
595     }
596
597     // ParseForm parses the raw query from the URL.
598     //
599     // For POST or PUT requests, it also parses the request body
600     // If the request Body's size has not already been limited b
601     // the size is capped at 10MB.
602     //
603     // ParseMultipartForm calls ParseForm automatically.
604     // It is idempotent.
605     func (r *Request) ParseForm() (err error) {
606         if r.Form != nil {
607             return
608         }
609         if r.URL != nil {
610             r.Form, err = url.ParseQuery(r.URL.RawQuery)
611         }
612         if r.Method == "POST" || r.Method == "PUT" {
613             if r.Body == nil {
614                 return errors.New("missing form body")
615             }
616             ct := r.Header.Get("Content-Type")
617             ct, _, err = mime.ParseMediaType(ct)
618             switch {
619             case ct == "application/x-www-form-urlencoded":
620                 var reader io.Reader = r.Body
621                 maxFormSize := int64(1<<63 - 1)
622                 if _, ok := r.Body.(*maxBytesReader) {
623                     maxFormSize = int64(10 << 20)
624                     reader = io.LimitReader(r.Body, maxFormSize)
625                 }
626                 b, e := ioutil.ReadAll(reader)
627                 if e != nil {
628                     if err == nil {
629                         err = e
630                     }
631                     break
632                 }
633                 if int64(len(b)) > maxFormSize {
634                     return errors.New("http: POST body too large")
635                 }
636                 var newValues url.Values

```

```

637         newValues, e = url.ParseQuery(string
638         if err == nil {
639             err = e
640         }
641         if r.Form == nil {
642             r.Form = make(url.Values)
643         }
644         // Copy values into r.Form. TODO: ma
645         for k, vs := range newValues {
646             for _, value := range vs {
647                 r.Form.Add(k, value)
648             }
649         }
650         case ct == "multipart/form-data":
651             // handled by ParseMultipartForm (wh
652             // TODO(bradfitz): there are too man
653             // orders to call too many functions
654             // Clean this up and write more test
655             // request_test.go contains the star
656             // in TestRequestMultipartCallOrder.
657         }
658     }
659     return err
660 }
661
662 // ParseMultipartForm parses a request body as multipart/form
663 // The whole request body is parsed and up to a total of max
664 // its file parts are stored in memory, with the remainder s
665 // disk in temporary files.
666 // ParseMultipartForm calls ParseForm if necessary.
667 // After one call to ParseMultipartForm, subsequent calls ha
668 func (r *Request) ParseMultipartForm(maxMemory int64) error
669     if r.MultipartForm == multipartByReader {
670         return errors.New("http: multipart handled b
671     }
672     if r.Form == nil {
673         err := r.ParseForm()
674         if err != nil {
675             return err
676         }
677     }
678     if r.MultipartForm != nil {
679         return nil
680     }
681
682     mr, err := r.multipartReader()
683     if err == ErrNotMultipart {
684         return nil
685     } else if err != nil {
686         return err

```

```

687     }
688
689     f, err := mr.ReadForm(maxMemory)
690     if err != nil {
691         return err
692     }
693     for k, v := range f.Value {
694         r.Form[k] = append(r.Form[k], v...)
695     }
696     r.MultipartForm = f
697
698     return nil
699 }
700
701 // FormValue returns the first value for the named component
702 // FormValue calls ParseMultipartForm and ParseForm if neces
703 func (r *Request) FormValue(key string) string {
704     if r.Form == nil {
705         r.ParseMultipartForm(defaultMaxMemory)
706     }
707     if vs := r.Form[key]; len(vs) > 0 {
708         return vs[0]
709     }
710     return ""
711 }
712
713 // FormFile returns the first file for the provided form key
714 // FormFile calls ParseMultipartForm and ParseForm if necess
715 func (r *Request) FormFile(key string) (multipart.File, *mul
716     if r.MultipartForm == multipartByReader {
717         return nil, nil, errors.New("http: multipart
718     }
719     if r.MultipartForm == nil {
720         err := r.ParseMultipartForm(defaultMaxMemory
721         if err != nil {
722             return nil, nil, err
723         }
724     }
725     if r.MultipartForm != nil && r.MultipartForm.File !=
726         if fhs := r.MultipartForm.File[key]; len(fhs
727             f, err := fhs[0].Open()
728             return f, fhs[0], err
729         }
730     }
731     return nil, nil, ErrMissingFile
732 }
733
734 func (r *Request) expectsContinue() bool {
735     return strings.ToLower(r.Header.Get("Expect")) == "1

```

```
736 }
737
738 func (r *Request) wantsHttp10KeepAlive() bool {
739     if r.ProtoMajor != 1 || r.ProtoMinor != 0 {
740         return false
741     }
742     return strings.Contains(strings.ToLower(r.Header.Get
743 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/response.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // HTTP Response reading and parsing.
6
7 package http
8
9 import (
10     "bufio"
11     "errors"
12     "io"
13     "net/textproto"
14     "net/url"
15     "strconv"
16     "strings"
17 )
18
19 var respExcludeHeader = map[string]bool{
20     "Content-Length": true,
21     "Transfer-Encoding": true,
22     "Trailer": true,
23 }
24
25 // Response represents the response from an HTTP request.
26 //
27 type Response struct {
28     Status      string // e.g. "200 OK"
29     StatusCode  int    // e.g. 200
30     Proto       string // e.g. "HTTP/1.0"
31     ProtoMajor  int    // e.g. 1
32     ProtoMinor  int    // e.g. 0
33
34     // Header maps header keys to values. If the respon
35     // headers with the same key, they will be concatena
36     // delimiters. (Section 4.2 of RFC 2616 requires th
37     // be semantically equivalent to a comma-delimited s
38     // duplicated by other fields in this struct (e.g.,
39     // omitted from Header.
40     //
41     // Keys in the map are canonicalized (see CanonicalH
```

```

42         Header Header
43
44         // Body represents the response body.
45         //
46         // The http Client and Transport guarantee that Body
47         // non-nil, even on responses without a body or resp
48         // a zero-lengthed body.
49         Body io.ReadCloser
50
51         // ContentLength records the length of the associate
52         // value -1 indicates that the length is unknown. U
53         // is "HEAD", values >= 0 indicate that the given nu
54         // be read from Body.
55         ContentLength int64
56
57         // Contains transfer encodings from outer-most to in
58         // nil, means that "identity" encoding is used.
59         TransferEncoding []string
60
61         // Close records whether the header directed that th
62         // closed after reading Body. The value is advice f
63         // ReadResponse nor Response.Write ever closes a con
64         Close bool
65
66         // Trailer maps trailer keys to values, in the same
67         // format as the header.
68         Trailer Header
69
70         // The Request that was sent to obtain this Response
71         // Request's Body is nil (having already been consum
72         // This is only populated for Client requests.
73         Request *Request
74     }
75
76     // Cookies parses and returns the cookies set in the Set-Coo
77     func (r *Response) Cookies() []*Cookie {
78         return readSetCookies(r.Header)
79     }
80
81     var ErrNoLocation = errors.New("http: no Location header in
82
83     // Location returns the URL of the response's "Location" hea
84     // if present. Relative redirects are resolved relative to
85     // the Response's Request. ErrNoLocation is returned if no
86     // Location header is present.
87     func (r *Response) Location() (*url.URL, error) {
88         lv := r.Header.Get("Location")
89         if lv == "" {
90             return nil, ErrNoLocation
91         }

```

```

92         if r.Request != nil && r.Request.URL != nil {
93             return r.Request.URL.Parse(lv)
94         }
95         return url.Parse(lv)
96     }
97
98     // ReadResponse reads and returns an HTTP response from r.
99     // req parameter specifies the Request that corresponds to
100    // this Response. Clients must call resp.Body.Close when fi
101    // reading resp.Body. After that call, clients can inspect
102    // resp.Trailer to find key/value pairs included in the resp
103    // trailer.
104    func ReadResponse(r *bufio.Reader, req *Request) (resp *Resp
105
106        tp := textproto.NewReader(r)
107        resp = new(Response)
108
109        resp.Request = req
110        resp.Request.Method = strings.ToUpper(resp.Request.M
111
112        // Parse the first line of the response.
113        line, err := tp.ReadLine()
114        if err != nil {
115            if err == io.EOF {
116                err = io.ErrUnexpectedEOF
117            }
118            return nil, err
119        }
120        f := strings.SplitN(line, " ", 3)
121        if len(f) < 2 {
122            return nil, &badStringError{"malformed HTTP
123        }
124        reasonPhrase := ""
125        if len(f) > 2 {
126            reasonPhrase = f[2]
127        }
128        resp.Status = f[1] + " " + reasonPhrase
129        resp.StatusCode, err = strconv.Atoi(f[1])
130        if err != nil {
131            return nil, &badStringError{"malformed HTTP
132        }
133
134        resp.Proto = f[0]
135        var ok bool
136        if resp.ProtoMajor, resp.ProtoMinor, ok = ParseHTTPV
137            return nil, &badStringError{"malformed HTTP
138        }
139
140        // Parse the response headers.

```

```

141         mimeHeader, err := tp.ReadMIMEHeader()
142         if err != nil {
143             return nil, err
144         }
145         resp.Header = Header(mimeHeader)
146
147         fixPragmaCacheControl(resp.Header)
148
149         err = readTransfer(resp, r)
150         if err != nil {
151             return nil, err
152         }
153
154         return resp, nil
155     }
156
157     // RFC2616: Should treat
158     //     Pragma: no-cache
159     // like
160     //     Cache-Control: no-cache
161     func fixPragmaCacheControl(header Header) {
162         if hp, ok := header["Pragma"]; ok && len(hp) > 0 &&
163             if _, presentcc := header["Cache-Control"];
164                 header["Cache-Control"] = []string{"
165             }
166     }
167 }
168
169 // ProtoAtLeast returns whether the HTTP protocol used
170 // in the response is at least major.minor.
171 func (r *Response) ProtoAtLeast(major, minor int) bool {
172     return r.ProtoMajor > major ||
173         r.ProtoMajor == major && r.ProtoMinor >= min
174 }
175
176 // Writes the response (header, body and trailer) in wire fo
177 // consults the following fields of the response:
178 //
179 //     StatusCode
180 //     ProtoMajor
181 //     ProtoMinor
182 //     RequestMethod
183 //     TransferEncoding
184 //     Trailer
185 //     Body
186 //     ContentLength
187 //     Header, values for non-canonical keys will have unpredic
188 //
189 func (r *Response) Write(w io.Writer) error {

```

```

190
191 // RequestMethod should be upper-case
192 if r.Request != nil {
193     r.Request.Method = strings.ToUpper(r.Request
194 }
195
196 // Status line
197 text := r.Status
198 if text == "" {
199     var ok bool
200     text, ok = statusText[r.StatusCode]
201     if !ok {
202         text = "status code " + strconv.Itoa
203     }
204 }
205 io.WriteString(w, "HTTP/"+strconv.Itoa(r.ProtoMajor)
206 io.WriteString(w, strconv.Itoa(r.ProtoMinor)+" ")
207 io.WriteString(w, strconv.Itoa(r.StatusCode)+" "+tex
208
209 // Process Body,ContentLength,Close,Trailer
210 tw, err := newTransferWriter(r)
211 if err != nil {
212     return err
213 }
214 err = tw.WriteHeader(w)
215 if err != nil {
216     return err
217 }
218
219 // Rest of header
220 err = r.Header.WriteSubset(w, respExcludeHeader)
221 if err != nil {
222     return err
223 }
224
225 // End-of-header
226 io.WriteString(w, "\r\n")
227
228 // Write body and trailer
229 err = tw.WriteBody(w)
230 if err != nil {
231     return err
232 }
233
234 // Success
235 return nil
236 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/server.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // HTTP server. See RFC 2616.
6
7 // TODO(rsc):
8 //     logging
9
10 package http
11
12 import (
13     "bufio"
14     "bytes"
15     "crypto/tls"
16     "errors"
17     "fmt"
18     "io"
19     "io/ioutil"
20     "log"
21     "net"
22     "net/url"
23     "path"
24     "runtime/debug"
25     "strconv"
26     "strings"
27     "sync"
28     "time"
29 )
30
31 // Errors introduced by the HTTP server.
32 var (
33     ErrWriteAfterFlush = errors.New("Conn.Write called a
34     ErrBodyNotAllowed = errors.New("http: response stat
35     ErrHijacked       = errors.New("Conn has been hijac
36     ErrContentLength  = errors.New("Conn.Write wrote mc
37 )
38
39 // Objects implementing the Handler interface can be
40 // registered to serve a particular path or subtree
41 // in the HTTP server.
42 //
43 // ServeHTTP should write reply headers and data to the Resp
44 // and then return. Returning signals that the request is f
```

```

45 // and that the HTTP server can move on to the next request
46 // the connection.
47 type Handler interface {
48     ServeHTTP(ResponseWriter, *Request)
49 }
50
51 // A ResponseWriter interface is used by an HTTP handler to
52 // construct an HTTP response.
53 type ResponseWriter interface {
54     // Header returns the header map that will be sent b
55     // Changing the header after a call to WriteHeader (
56     // no effect.
57     Header() Header
58
59     // Write writes the data to the connection as part o
60     // If WriteHeader has not yet been called, Write cal
61     // before writing the data. If the Header does not
62     // Content-Type line, Write adds a Content-Type set
63     // the initial 512 bytes of written data to DetectCo
64     Write([]byte) (int, error)
65
66     // WriteHeader sends an HTTP response header with st
67     // If WriteHeader is not called explicitly, the firs
68     // will trigger an implicit WriteHeader(http.StatusC
69     // Thus explicit calls to WriteHeader are mainly use
70     // send error codes.
71     WriteHeader(int)
72 }
73
74 // The Flusher interface is implemented by ResponseWriters t
75 // an HTTP handler to flush buffered data to the client.
76 //
77 // Note that even for ResponseWriters that support Flush,
78 // if the client is connected through an HTTP proxy,
79 // the buffered data may not reach the client until the resp
80 // completes.
81 type Flusher interface {
82     // Flush sends any buffered data to the client.
83     Flush()
84 }
85
86 // The Hijacker interface is implemented by ResponseWriters
87 // an HTTP handler to take over the connection.
88 type Hijacker interface {
89     // Hijack lets the caller take over the connection.
90     // After a call to Hijack(), the HTTP server library
91     // will not do anything else with the connection.
92     // It becomes the caller's responsibility to manage
93     // and close the connection.
94     Hijack() (net.Conn, *bufio.ReadWriter, error)

```

```

95 }
96
97 // A conn represents the server side of an HTTP connection.
98 type conn struct {
99     remoteAddr string           // network address o
100    server      *Server           // the Server on whi
101    rwc         net.Conn          // i/o connection
102    lr         *io.LimitedReader // io.LimitReader(rw
103    buf        *bufio.ReadWriter // buffered(lr, rwc),
104    hijacked   bool             // connection has be
105    tlsState   *tls.ConnectionState // or nil when not u
106    body       []byte
107 }
108
109 // A response represents the server side of an HTTP response
110 type response struct {
111     conn      *conn
112     req       *Request // request for this response
113     chunking  bool     // using chunked transfer enc
114     wroteHeader bool    // reply header has been writ
115     wroteContinue bool // 100 Continue response was
116     header    Header  // reply header parameters
117     written   int64   // number of bytes written in
118     contentLength int64 // explicitly-declared Conten
119     status    int     // status code passed to Writ
120     needSniff bool    // need to sniff to find Cont
121
122     // close connection after this reply. set on reques
123     // updated after response from handler if there's a
124     // "Connection: keep-alive" response header and a
125     // Content-Length.
126     closeAfterReply bool
127
128     // requestBodyLimitHit is set by requestTooLarge whe
129     // maxBytesReader hits its max size. It is checked i
130     // WriteHeader, to make sure we don't consume the th
131     // remaining request body to try to advance to the n
132     // request. Instead, when this is set, we stop doing
133     // subsequent requests on this connection and stop r
134     // input from it.
135     requestBodyLimitHit bool
136 }
137
138 // requestTooLarge is called by maxBytesReader when too much
139 // been read from the client.
140 func (w *response) requestTooLarge() {
141     w.closeAfterReply = true
142     w.requestBodyLimitHit = true
143     if !w.wroteHeader {

```

```

144             w.Header().Set("Connection", "close")
145         }
146     }
147
148     type writerOnly struct {
149         io.Writer
150     }
151
152     func (w *response) ReadFrom(src io.Reader) (n int64, err error) {
153         // Call WriteHeader before checking w.chunking if it
154         // been called yet, since WriteHeader is what sets w
155         if !w.wroteHeader {
156             w.WriteHeader(StatusOK)
157         }
158         if !w.chunking && w.bodyAllowed() && !w.needSniff {
159             w.Flush()
160             if rf, ok := w.conn.rwc.(io.ReaderFrom); ok {
161                 n, err = rf.ReadFrom(src)
162                 w.written += n
163                 return
164             }
165         }
166         // Fall back to default io.Copy implementation.
167         // Use wrapper to hide w.ReadFrom from io.Copy.
168         return io.Copy(writerOnly{w}, src)
169     }
170
171     // noLimit is an effective infinite upper bound for io.Limit
172     const noLimit int64 = (1 << 63) - 1
173
174     // Create new connection from rwc.
175     func (srv *Server) newConn(rwc net.Conn) (c *conn, err error) {
176         c = new(conn)
177         c.remoteAddr = rwc.RemoteAddr().String()
178         c.server = srv
179         c.rwc = rwc
180         c.body = make([]byte, sniffLen)
181         c.lr = io.LimitReader(rwc, noLimit).(*io.LimitedReader)
182         br := bufio.NewReader(c.lr)
183         bw := bufio.NewWriter(rwc)
184         c.buf = bufio.NewReadWriter(br, bw)
185         return c, nil
186     }
187
188     // DefaultMaxHeaderBytes is the maximum permitted size of th
189     // in an HTTP request.
190     // This can be overridden by setting Server.MaxHeaderBytes.
191     const DefaultMaxHeaderBytes = 1 << 20 // 1 MB
192

```

```

193 func (srv *Server) maxHeaderBytes() int {
194     if srv.MaxHeaderBytes > 0 {
195         return srv.MaxHeaderBytes
196     }
197     return DefaultMaxHeaderBytes
198 }
199
200 // wrapper around io.ReaderCloser which on first read, sends
201 // HTTP/1.1 100 Continue header
202 type expectContinueReader struct {
203     resp      *response
204     readCloser io.ReadCloser
205     closed    bool
206 }
207
208 func (ecr *expectContinueReader) Read(p []byte) (n int, err
209     if ecr.closed {
210         return 0, errors.New("http: Read after Close
211     }
212     if !ecr.resp.wroteContinue && !ecr.resp.conn.hijacked
213         ecr.resp.wroteContinue = true
214         io.WriteString(ecr.resp.conn.buf, "HTTP/1.1
215         ecr.resp.conn.buf.Flush()
216     }
217     return ecr.readCloser.Read(p)
218 }
219
220 func (ecr *expectContinueReader) Close() error {
221     ecr.closed = true
222     return ecr.readCloser.Close()
223 }
224
225 // TimeFormat is the time format to use with
226 // time.Parse and time.Time.Format when parsing
227 // or generating times in HTTP headers.
228 // It is like time.RFC1123 but hard codes GMT as the time zone
229 const TimeFormat = "Mon, 02 Jan 2006 15:04:05 GMT"
230
231 var errTooLarge = errors.New("http: request too large")
232
233 // Read next request from connection.
234 func (c *conn) readRequest() (w *response, err error) {
235     if c.hijacked {
236         return nil, ErrHijacked
237     }
238     c.lr.N = int64(c.server.maxHeaderBytes()) + 4096 /*
239     var req *Request
240     if req, err = ReadRequest(c.buf.Reader); err != nil
241         if c.lr.N == 0 {
242         return nil, errTooLarge

```

```

243         }
244         return nil, err
245     }
246     c.lr.N = noLimit
247
248     req.RemoteAddr = c.remoteAddr
249     req.TLS = c.tlsState
250
251     w = new(response)
252     w.conn = c
253     w.req = req
254     w.header = make(Header)
255     w.contentLength = -1
256     c.body = c.body[:0]
257     return w, nil
258 }
259
260 func (w *response) Header() Header {
261     return w.header
262 }
263
264 // maxPostHandlerReadBytes is the max number of Request.Body
265 // consumed by a handler that the server will read from the
266 // in order to keep a connection alive. If there are more b
267 // this then the server to be paranoid instead sends a "Conn
268 // close" response.
269 //
270 // This number is approximately what a typical machine's TCP
271 // size is anyway. (if we have the bytes on the machine, we
272 // well read them)
273 const maxPostHandlerReadBytes = 256 << 10
274
275 func (w *response) WriteHeader(code int) {
276     if w.conn.hijacked {
277         log.Print("http: response.WriteHeader on hij
278             return
279     }
280     if w.wroteHeader {
281         log.Print("http: multiple response.WriteHeader
282             return
283     }
284     w.wroteHeader = true
285     w.status = code
286
287     // Check for a explicit (and valid) Content-Length h
288     var hasCL bool
289     var contentLength int64
290     if clenStr := w.header.Get("Content-Length"); clenSt
291         var err error

```

```

292         contentLength, err = strconv.ParseInt(clenSt
293         if err == nil {
294             hasCL = true
295         } else {
296             log.Printf("http: invalid Content-Le
297             w.header.Del("Content-Length")
298         }
299     }
300
301     if w.req.wantsHttp10KeepAlive() && (w.req.Method ==
302     _, connectionHeaderSet := w.header["Connecti
303     if !connectionHeaderSet {
304         w.header.Set("Connection", "keep-ali
305     }
306 } else if !w.req.ProtoAtLeast(1, 1) {
307     // Client did not ask to keep connection ali
308     w.closeAfterReply = true
309 }
310
311 if w.header.Get("Connection") == "close" {
312     w.closeAfterReply = true
313 }
314
315 // Per RFC 2616, we should consume the request body
316 // replying, if the handler hasn't already done so.
317 // don't want to do an unbounded amount of reading h
318 // DoS reasons, so we only try up to a threshold.
319 if w.req.ContentLength != 0 && !w.closeAfterReply {
320     ecr, isExpecter := w.req.Body.(*expectContin
321     if !isExpecter || ecr.resp.wroteContinue {
322         n, _ := io.CopyN(ioutil.Discard, w.r
323         if n >= maxPostHandlerReadBytes {
324             w.requestTooLarge()
325             w.header.Set("Connection", "
326         } else {
327             w.req.Body.Close()
328         }
329     }
330 }
331
332 if code == StatusNotModified {
333     // Must not have body.
334     for _, header := range []string{"Content-Typ
335         if w.header.Get(header) != "" {
336             // TODO: return an error if
337             // or set a flag on w to mak
338             // for now just log and drop
339             log.Printf("http: StatusNotM
340             w.header.Del(header)

```

```

341         }
342     }
343 } else {
344     // If no content type, apply sniffing algorithm
345     if w.header.Get("Content-Type") == "" && w.req.Header().Get("Content-Type") == "" {
346         w.needSniff = true
347     }
348 }
349
350 if _, ok := w.header["Date"]; !ok {
351     w.Header().Set("Date", time.Now().UTC().Format("Mon, 2 Jan 2006 15:04:05 -0700"))
352 }
353
354 te := w.header.Get("Transfer-Encoding")
355 hasTE := te != ""
356 if hasCL && hasTE && te != "identity" {
357     // TODO: return an error if WriteHeader gets called with both Content-Length and
358     // Transfer-Encoding. For now just ignore the Content-Length.
359     log.Printf("http: WriteHeader called with both Content-Length and Transfer-Encoding (%s)", te)
360     w.header.Del("Content-Length")
361     hasCL = false
362 }
363
364 if w.req.Method == "HEAD" || code == StatusNotModified {
365     // do nothing
366 } else if hasCL {
367     w.contentLength = contentLength
368     w.header.Del("Transfer-Encoding")
369 } else if w.req.ProtoAtLeast(1, 1) {
370     // HTTP/1.1 or greater: use chunked transfer encoding to avoid closing the connection at EOF.
371     // TODO: this blows away any custom or standard Content-Length header that the user
372     // might have set. Deal with that as needed in a future version.
373     // use case.
374     w.chunking = true
375     w.header.Set("Transfer-Encoding", "chunked")
376 } else {
377     // HTTP version < 1.1: cannot do chunked transfer encoding and we don't know the Content-Length.
378     // signal EOF by closing connection.
379     w.closeAfterReply = true
380     w.header.Del("Transfer-Encoding") // in case it was set
381 }
382
383 // Cannot use Content-Length with non-identity Transfer-Encoding
384 if w.chunking {
385     w.header.Del("Content-Length")
386 }
387
388 if !w.req.ProtoAtLeast(1, 0) {

```

```

391         return
392     }
393     proto := "HTTP/1.0"
394     if w.req.ProtoAtLeast(1, 1) {
395         proto = "HTTP/1.1"
396     }
397     codestring := strconv.Itoa(code)
398     text, ok := statusText[code]
399     if !ok {
400         text = "status code " + codestring
401     }
402     io.WriteString(w.conn.buf, proto+" "+codestring+" "+
403     w.header.Write(w.conn.buf)
404
405     // If we need to sniff the body, leave the header op
406     // Otherwise, end it here.
407     if !w.needSniff {
408         io.WriteString(w.conn.buf, "\r\n")
409     }
410 }
411
412 // sniff uses the first block of written data,
413 // stored in w.conn.body, to decide the Content-Type
414 // for the HTTP body.
415 func (w *response) sniff() {
416     if !w.needSniff {
417         return
418     }
419     w.needSniff = false
420
421     data := w.conn.body
422     fmt.Fprintf(w.conn.buf, "Content-Type: %s\r\n\r\n",
423
424     if len(data) == 0 {
425         return
426     }
427     if w.chunking {
428         fmt.Fprintf(w.conn.buf, "%x\r\n", len(data))
429     }
430     _, err := w.conn.buf.Write(data)
431     if w.chunking && err == nil {
432         io.WriteString(w.conn.buf, "\r\n")
433     }
434 }
435
436 // bodyAllowed returns true if a Write is allowed for this r
437 // It's illegal to call this before the header has been flush
438 func (w *response) bodyAllowed() bool {
439     if !w.wroteHeader {

```

```

440         panic("")
441     }
442     return w.status != StatusNotModified && w.req.Method
443 }
444
445 func (w *response) Write(data []byte) (n int, err error) {
446     if w.conn.hijacked {
447         log.Print("http: response.Write on hijacked")
448         return 0, ErrHijacked
449     }
450     if !w.wroteHeader {
451         w.WriteHeader(StatusOK)
452     }
453     if len(data) == 0 {
454         return 0, nil
455     }
456     if !w.bodyAllowed() {
457         return 0, ErrBodyNotAllowed
458     }
459
460     w.written += int64(len(data)) // ignoring errors, fo
461     if w.contentType != -1 && w.written > w.contentType
462         return 0, ErrContentLength
463     }
464
465     var m int
466     if w.needSniff {
467         // We need to sniff the beginning of the out
468         // determine the content type. Accumulate t
469         // initial writes in w.conn.body.
470         // Cap m so that append won't allocate.
471         m = cap(w.conn.body) - len(w.conn.body)
472         if m > len(data) {
473             m = len(data)
474         }
475         w.conn.body = append(w.conn.body, data[:m]..
476         data = data[m:]
477         if len(data) == 0 {
478             // Copied everything into the buffer
479             // Wait for next write.
480             return m, nil
481         }
482
483         // Filled the buffer; more data remains.
484         // Sniff the content (flushes the buffer)
485         // and then proceed with the remainder
486         // of the data as a normal Write.
487         // Calling sniff clears needSniff.
488         w.sniff()

```

```

489     }
490
491     // TODO(rsc): if chunking happened after the bufferi
492     // then there would be fewer chunk headers.
493     // On the other hand, it would make hijacking more d
494     if w.chunking {
495         fmt.Fprintf(w.conn.buf, "%x\r\n", len(data))
496     }
497     n, err = w.conn.buf.Write(data)
498     if err == nil && w.chunking {
499         if n != len(data) {
500             err = io.ErrShortWrite
501         }
502         if err == nil {
503             io.WriteString(w.conn.buf, "\r\n")
504         }
505     }
506
507     return m + n, err
508 }
509
510 func (w *response) finishRequest() {
511     // If this was an HTTP/1.0 request with keep-alive a
512     // back, we can make this a keep-alive response ...
513     if w.req.wantsHttp10KeepAlive() {
514         sentLength := w.header.Get("Content-Length")
515         if sentLength && w.header.Get("Connection")
516             w.closeAfterReply = false
517     }
518 }
519 if !w.wroteHeader {
520     w.WriteHeader(StatusOK)
521 }
522 if w.needSniff {
523     w.sniff()
524 }
525 if w.chunking {
526     io.WriteString(w.conn.buf, "0\r\n")
527     // trailer key/value pairs, followed by blan
528     io.WriteString(w.conn.buf, "\r\n")
529 }
530 w.conn.buf.Flush()
531 // Close the body, unless we're about to close the w
532 // anyway.
533 if !w.closeAfterReply {
534     w.req.Body.Close()
535 }
536 if w.req.MultipartForm != nil {
537     w.req.MultipartForm.RemoveAll()
538 }

```

```

539
540     if w.contentLength != -1 && w.contentLength != w.wri
541         // Did not write enough. Avoid getting out o
542         w.closeAfterReply = true
543     }
544 }
545
546 func (w *response) Flush() {
547     if !w.wroteHeader {
548         w.WriteHeader(StatusOK)
549     }
550     w.sniff()
551     w.conn.buf.Flush()
552 }
553
554 // Close the connection.
555 func (c *conn) close() {
556     if c.buf != nil {
557         c.buf.Flush()
558         c.buf = nil
559     }
560     if c.rwc != nil {
561         c.rwc.Close()
562         c.rwc = nil
563     }
564 }
565
566 // Serve a new connection.
567 func (c *conn) serve() {
568     defer func() {
569         err := recover()
570         if err == nil {
571             return
572         }
573
574         var buf bytes.Buffer
575         fmt.Fprintf(&buf, "http: panic serving %v: %
576         buf.Write(debug.Stack())
577         log.Print(buf.String())
578
579         if c.rwc != nil { // may be nil if connectio
580             c.rwc.Close()
581         }
582     }()
583
584     if tlsConn, ok := c.rwc.(*tls.Conn); ok {
585         if err := tlsConn.Handshake(); err != nil {
586             c.close()
587             return

```

```

588     }
589     c.tlsState = new(tls.ConnectionState)
590     *c.tlsState = tlsConn.ConnectionState()
591 }
592
593 for {
594     w, err := c.readRequest()
595     if err != nil {
596         msg := "400 Bad Request"
597         if err == errTooLarge {
598             // Their HTTP client may or
599             // able to read this if we'r
600             // responding to them and ha
601             // while they're still writi
602             // request. Undefined behav
603             msg = "413 Request Entity To
604         } else if err == io.EOF {
605             break // Don't reply
606         } else if neterr, ok := err.(net.Err
607             break // Don't reply
608         }
609         fmt.Fprintf(c.rwc, "HTTP/1.1 %s\r\n\
610         break
611     }
612
613     // Expect 100 Continue support
614     req := w.req
615     if req.expectsContinue() {
616         if req.ProtoAtLeast(1, 1) {
617             // Wrap the Body reader with
618             req.Body = &expectContinueRe
619         }
620         if req.ContentLength == 0 {
621             w.Header().Set("Connection",
622             w.WriteHeader(StatusBadReque
623             w.finishRequest()
624             break
625         }
626         req.Header.Del("Expect")
627     } else if req.Header.Get("Expect") != "" {
628         // TODO(bradfitz): let ServeHTTP han
629         // requests with non-standard expect
630         // theoretical at best, and doesn't
631         // current ServeHTTP model anyway.
632         // make the ResponseWriter an option
633         // "ExpectReplier" interface or some
634         //
635         // For now we'll just obey RFC 2616
636         // "If a server receives a request c

```

```

637             // Expect field that includes an exp
638             // extension that it does not support
639             // respond with a 417 (Expectation F
640             w.Header().Set("Connection", "close"
641             w.WriteHeader(StatusExpectationFaile
642             w.finishRequest()
643             break
644         }
645
646         handler := c.server.Handler
647         if handler == nil {
648             handler = DefaultServeMux
649         }
650
651         // HTTP cannot have multiple simultaneous ac
652         // Until the server replies to this request,
653         // so we might as well run the handler in th
654         // [*] Not strictly true: HTTP pipelining.
655         // in parallel even if their responses need
656         handler.ServeHTTP(w, w.req)
657         if c.hijacked {
658             return
659         }
660         w.finishRequest()
661         if w.closeAfterReply {
662             break
663         }
664     }
665     c.close()
666 }
667
668 // Hijack implements the Hijacker.Hijack method. Our respons
669 // and a Hijacker.
670 func (w *response) Hijack() (rwc net.Conn, buf *bufio.ReadWr
671     if w.conn.hijacked {
672         return nil, nil, ErrHijacked
673     }
674     w.conn.hijacked = true
675     rwc = w.conn.rwc
676     buf = w.conn.buf
677     w.conn.rwc = nil
678     w.conn.buf = nil
679     return
680 }
681
682 // The HandlerFunc type is an adapter to allow the use of
683 // ordinary functions as HTTP handlers. If f is a function
684 // with the appropriate signature, HandlerFunc(f) is a
685 // Handler object that calls f.
686 type HandlerFunc func(ResponseWriter, *Request)

```

```

687
688 // ServeHTTP calls f(w, r).
689 func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request)
690     f(w, r)
691 }
692
693 // Helper handlers
694
695 // Error replies to the request with the specified error mes
696 func Error(w ResponseWriter, error string, code int) {
697     w.Header().Set("Content-Type", "text/plain; charset="
698     w.WriteHeader(code)
699     fmt.Fprintln(w, error)
700 }
701
702 // NotFound replies to the request with an HTTP 404 not four
703 func NotFound(w ResponseWriter, r *Request) { Error(w, "404
704
705 // NotFoundHandler returns a simple request handler
706 // that replies to each request with a ``404 page not found'
707 func NotFoundHandler() Handler { return HandlerFunc(NotFound
708
709 // StripPrefix returns a handler that serves HTTP requests
710 // by removing the given prefix from the request URL's Path
711 // and invoking the handler h. StripPrefix handles a
712 // request for a path that doesn't begin with prefix by
713 // replying with an HTTP 404 not found error.
714 func StripPrefix(prefix string, h Handler) Handler {
715     return HandlerFunc(func(w ResponseWriter, r *Request
716         if !strings.HasPrefix(r.URL.Path, prefix) {
717             NotFound(w, r)
718             return
719         }
720         r.URL.Path = r.URL.Path[len(prefix):]
721         h.ServeHTTP(w, r)
722     })
723 }
724
725 // Redirect replies to the request with a redirect to url,
726 // which may be a path relative to the request path.
727 func Redirect(w ResponseWriter, r *Request, urlStr string, c
728     if u, err := url.Parse(urlStr); err == nil {
729         // If url was relative, make absolute by
730         // combining with request path.
731         // The browser would probably do this for us
732         // but doing it ourselves is more reliable.
733
734         // NOTE(rsc): RFC 2616 says that the Locatio
735         // line must be an absolute URI, like

```

```

736 // "http://www.google.com/redirect/",
737 // not a path like "/redirect/".
738 // Unfortunately, we don't know what to
739 // put in the host name section to get the
740 // client to connect to us again, so we can't
741 // know the right absolute URI to send back.
742 // Because of this problem, no one pays atte
743 // to the RFC; they all send back just a new
744 // So do we.
745 oldpath := r.URL.Path
746 if oldpath == "" { // should not happen, but
747     oldpath = "/"
748 }
749 if u.Scheme == "" {
750     // no leading http://server
751     if urlStr == "" || urlStr[0] != '/'
752         // make relative path absolu
753         olddir, _ := path.Split(oldp
754         urlStr = olddir + urlStr
755     }
756
757     var query string
758     if i := strings.Index(urlStr, "?");
759         urlStr, query = urlStr[:i],
760     }
761
762     // clean up but preserve trailing sl
763     trailing := urlStr[len(urlStr)-1] ==
764     urlStr = path.Clean(urlStr)
765     if trailing && urlStr[len(urlStr)-1]
766         urlStr += "/"
767     }
768     urlStr += query
769 }
770 }
771
772 w.Header().Set("Location", urlStr)
773 w.WriteHeader(code)
774
775 // RFC2616 recommends that a short note "SHOULD" be
776 // response because older user agents may not unders
777 // Shouldn't send the response for POST or HEAD; tha
778 if r.Method == "GET" {
779     note := "<a href=\"\" + htmlEscape(urlStr) +
780     fmt.Fprintln(w, note)
781 }
782 }
783
784 var htmlReplacer = strings.NewReplacer(

```

```

785     "&", "&";",
786     "<", "&lt;";",
787     ">", "&gt;";",
788     // """ is shorter than "&quot;".
789     "`", """,
790     // "'" is shorter than "&apos;" and apos was not
791     "'", "'",
792 )
793
794 func htmlEscape(s string) string {
795     return htmlReplacer.Replace(s)
796 }
797
798 // Redirect to a fixed URL
799 type redirectHandler struct {
800     url string
801     code int
802 }
803
804 func (rh *redirectHandler) ServeHTTP(w ResponseWriter, r *Re
805     Redirect(w, r, rh.url, rh.code)
806 }
807
808 // RedirectHandler returns a request handler that redirects
809 // each request it receives to the given url using the given
810 // status code.
811 func RedirectHandler(url string, code int) Handler {
812     return &redirectHandler{url, code}
813 }
814
815 // ServeMux is an HTTP request multiplexer.
816 // It matches the URL of each incoming request against a lis
817 // patterns and calls the handler for the pattern that
818 // most closely matches the URL.
819 //
820 // Patterns named fixed, rooted paths, like "/favicon.ico",
821 // or rooted subtrees, like "/images/" (note the trailing sl
822 // Longer patterns take precedence over shorter ones, so tha
823 // if there are handlers registered for both "/images/"
824 // and "/images/thumbnails/", the latter handler will be
825 // called for paths beginning "/images/thumbnails/" and the
826 // former will receiver requests for any other paths in the
827 // "/images/" subtree.
828 //
829 // Patterns may optionally begin with a host name, restricti
830 // URLs on that host only. Host-specific patterns take prec
831 // general patterns, so that a handler might register for th
832 // "/codesearch" and "codesearch.google.com/" without also t
833 // requests for "http://www.google.com/".
834 //

```

```

835 // ServeMux also takes care of sanitizing the URL request pa
836 // redirecting any request containing . or .. elements to an
837 // equivalent .- and ..-free URL.
838 type ServeMux struct {
839     mu sync.RWMutex
840     m map[string]muxEntry
841 }
842
843 type muxEntry struct {
844     explicit bool
845     h          Handler
846 }
847
848 // NewServeMux allocates and returns a new ServeMux.
849 func NewServeMux() *ServeMux { return &ServeMux{m: make(map[
850
851 // DefaultServeMux is the default ServeMux used by Serve.
852 var DefaultServeMux = NewServeMux()
853
854 // Does path match pattern?
855 func pathMatch(pattern, path string) bool {
856     if len(pattern) == 0 {
857         // should not happen
858         return false
859     }
860     n := len(pattern)
861     if pattern[n-1] != '/' {
862         return pattern == path
863     }
864     return len(path) >= n && path[0:n] == pattern
865 }
866
867 // Return the canonical path for p, eliminating . and .. ele
868 func cleanPath(p string) string {
869     if p == "" {
870         return "/"
871     }
872     if p[0] != '/' {
873         p = "/" + p
874     }
875     np := path.Clean(p)
876     // path.Clean removes trailing slash except for root
877     // put the trailing slash back if necessary.
878     if p[len(p)-1] == '/' && np != "/" {
879         np += "/"
880     }
881     return np
882 }
883

```

```

884 // Find a handler on a handler map given a path string
885 // Most-specific (longest) pattern wins
886 func (mux *ServeMux) match(path string) Handler {
887     var h Handler
888     var n = 0
889     for k, v := range mux.m {
890         if !pathMatch(k, path) {
891             continue
892         }
893         if h == nil || len(k) > n {
894             n = len(k)
895             h = v.h
896         }
897     }
898     return h
899 }
900
901 // handler returns the handler to use for the request r.
902 func (mux *ServeMux) handler(r *Request) Handler {
903     mux.mu.RLock()
904     defer mux.mu.RUnlock()
905
906     // Host-specific pattern takes precedence over gener
907     h := mux.match(r.Host + r.URL.Path)
908     if h == nil {
909         h = mux.match(r.URL.Path)
910     }
911     if h == nil {
912         h = NotFoundHandler()
913     }
914     return h
915 }
916
917 // ServeHTTP dispatches the request to the handler whose
918 // pattern most closely matches the request URL.
919 func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
920     // Clean path to canonical form and redirect.
921     if p := cleanPath(r.URL.Path); p != r.URL.Path {
922         w.Header().Set("Location", p)
923         w.WriteHeader(StatusMovedPermanently)
924         return
925     }
926     mux.handler(r).ServeHTTP(w, r)
927 }
928
929 // Handle registers the handler for the given pattern.
930 // If a handler already exists for pattern, Handle panics.
931 func (mux *ServeMux) Handle(pattern string, handler Handler)
932     mux.mu.Lock()

```



```

983 type Server struct {
984     Addr      string      // TCP address to liste
985     Handler   Handler     // handler to invoke, h
986     ReadTimeout  time.Duration // maximum duration bef
987     WriteTimeout time.Duration // maximum duration bef
988     MaxHeaderBytes int        // maximum size of requ
989     TLSConfig   *tls.Config // optional TLS config,
990 }
991
992 // ListenAndServe listens on the TCP network address srv.Addr
993 // calls Serve to handle requests on incoming connections.
994 // srv.Addr is blank, ":http" is used.
995 func (srv *Server) ListenAndServe() error {
996     addr := srv.Addr
997     if addr == "" {
998         addr = ":http"
999     }
1000    l, e := net.Listen("tcp", addr)
1001    if e != nil {
1002        return e
1003    }
1004    return srv.Serve(l)
1005 }
1006
1007 // Serve accepts incoming connections on the Listener l, cre
1008 // new service thread for each. The service threads read re
1009 // then call srv.Handler to reply to them.
1010 func (srv *Server) Serve(l net.Listener) error {
1011     defer l.Close()
1012     var tempDelay time.Duration // how long to sleep on
1013     for {
1014         rw, e := l.Accept()
1015         if e != nil {
1016             if ne, ok := e.(net.Error); ok && ne
1017                 if tempDelay == 0 {
1018                     tempDelay = 5 * time
1019                 } else {
1020                     tempDelay *= 2
1021                 }
1022                 if max := 1 * time.Second; t
1023                     tempDelay = max
1024                 }
1025                 log.Printf("http: Accept err
1026                 time.Sleep(tempDelay)
1027                 continue
1028             }
1029             return e
1030         }
1031         tempDelay = 0

```

```

1032         if srv.ReadTimeout != 0 {
1033             rw.SetReadDeadline(time.Now().Add(sr
1034         }
1035         if srv.WriteTimeout != 0 {
1036             rw.SetWriteDeadline(time.Now().Add(s
1037         }
1038         c, err := srv.newConn(rw)
1039         if err != nil {
1040             continue
1041         }
1042         go c.serve()
1043     }
1044     panic("not reached")
1045 }
1046
1047 // ListenAndServe listens on the TCP network address addr
1048 // and then calls Serve with handler to handle requests
1049 // on incoming connections. Handler is typically nil,
1050 // in which case the DefaultServeMux is used.
1051 //
1052 // A trivial example server is:
1053 //
1054 //     package main
1055 //
1056 //     import (
1057 //         "io"
1058 //         "net/http"
1059 //         "log"
1060 //     )
1061 //
1062 //     // hello world, the web server
1063 //     func HelloServer(w http.ResponseWriter, req *http.Re
1064 //         io.WriteString(w, "hello, world!\n")
1065 //     }
1066 //
1067 //     func main() {
1068 //         http.HandleFunc("/hello", HelloServer)
1069 //         err := http.ListenAndServe(":12345", nil)
1070 //         if err != nil {
1071 //             log.Fatal("ListenAndServe: ", err)
1072 //         }
1073 //     }
1074 func ListenAndServe(addr string, handler Handler) error {
1075     server := &Server{Addr: addr, Handler: handler}
1076     return server.ListenAndServe()
1077 }
1078
1079 // ListenAndServeTLS acts identically to ListenAndServe, exc
1080 // expects HTTPS connections. Additionally, files containing

```

```

1081 // matching private key for the server must be provided. If
1082 // is signed by a certificate authority, the certFile should
1083 // of the server's certificate followed by the CA's certific
1084 //
1085 // A trivial example server is:
1086 //
1087 //     import (
1088 //         "log"
1089 //         "net/http"
1090 //     )
1091 //
1092 //     func handler(w http.ResponseWriter, req *http.Reques
1093 //         w.Header().Set("Content-Type", "text/plain")
1094 //         w.Write([]byte("This is an example server.\n
1095 //     }
1096 //
1097 //     func main() {
1098 //         http.HandleFunc("/", handler)
1099 //         log.Printf("About to listen on 10443. Go to
1100 //         err := http.ListenAndServeTLS(":10443", "cer
1101 //         if err != nil {
1102 //             log.Fatal(err)
1103 //         }
1104 //     }
1105 //
1106 // One can use generate_cert.go in crypto/tls to generate ce
1107 func ListenAndServeTLS(addr string, certFile string, keyFile
1108     server := &Server{Addr: addr, Handler: handler}
1109     return server.ListenAndServeTLS(certFile, keyFile)
1110 }
1111
1112 // ListenAndServeTLS listens on the TCP network address srv.
1113 // then calls Serve to handle requests on incoming TLS conne
1114 //
1115 // Filenames containing a certificate and matching private k
1116 // the server must be provided. If the certificate is signed
1117 // certificate authority, the certFile should be the concate
1118 // of the server's certificate followed by the CA's certific
1119 //
1120 // If srv.Addr is blank, ":https" is used.
1121 func (srv *Server) ListenAndServeTLS(certFile, keyFile strin
1122     addr := srv.Addr
1123     if addr == "" {
1124         addr = ":https"
1125     }
1126     config := &tls.Config{}
1127     if srv.TLSConfig != nil {
1128         *config = *srv.TLSConfig
1129     }
1130     if config.NextProtos == nil {

```

```

1131         config.NextProtos = []string{"http/1.1"}
1132     }
1133
1134     var err error
1135     config.Certificates = make([]tls.Certificate, 1)
1136     config.Certificates[0], err = tls.LoadX509KeyPair(ce
1137     if err != nil {
1138         return err
1139     }
1140
1141     conn, err := net.Listen("tcp", addr)
1142     if err != nil {
1143         return err
1144     }
1145
1146     tlsListener := tls.NewListener(conn, config)
1147     return srv.Serve(tlsListener)
1148 }
1149
1150 // TimeoutHandler returns a Handler that runs h with the giv
1151 //
1152 // The new Handler calls h.ServeHTTP to handle each request,
1153 // call runs for more than ns nanoseconds, the handler respo
1154 // a 503 Service Unavailable error and the given message in
1155 // (If msg is empty, a suitable default message will be sent
1156 // After such a timeout, writes by h to its ResponseWriter w
1157 // ErrorHandlerTimeout.
1158 func TimeoutHandler(h Handler, dt time.Duration, msg string)
1159     f := func() <-chan time.Time {
1160         return time.After(dt)
1161     }
1162     return &timeoutHandler{h, f, msg}
1163 }
1164
1165 // ErrorHandlerTimeout is returned on ResponseWriter Write cal
1166 // in handlers which have timed out.
1167 var ErrorHandlerTimeout = errors.New("http: Handler timeout")
1168
1169 type timeoutHandler struct {
1170     handler Handler
1171     timeout func() <-chan time.Time // returns channel p
1172     body    string
1173 }
1174
1175 func (h *timeoutHandler) errorBody() string {
1176     if h.body != "" {
1177         return h.body
1178     }
1179     return "<html><head><title>Timeout</title></head><bo

```

```

1180 }
1181
1182 func (h *timeoutHandler) ServeHTTP(w ResponseWriter, r *Requ
1183     done := make(chan bool)
1184     tw := &timeoutWriter{w: w}
1185     go func() {
1186         h.handler.ServeHTTP(tw, r)
1187         done <- true
1188     }()
1189     select {
1190     case <-done:
1191         return
1192     case <-h.timeout():
1193         tw.mu.Lock()
1194         defer tw.mu.Unlock()
1195         if !tw.wroteHeader {
1196             tw.w.WriteHeader(StatusServiceUnavai
1197                 tw.w.Write([]byte(h.errorBody()))
1198         }
1199         tw.timedOut = true
1200     }
1201 }
1202
1203 type timeoutWriter struct {
1204     w ResponseWriter
1205
1206     mu          sync.Mutex
1207     timedOut    bool
1208     wroteHeader bool
1209 }
1210
1211 func (tw *timeoutWriter) Header() Header {
1212     return tw.w.Header()
1213 }
1214
1215 func (tw *timeoutWriter) Write(p []byte) (int, error) {
1216     tw.mu.Lock()
1217     timedOut := tw.timedOut
1218     tw.mu.Unlock()
1219     if timedOut {
1220         return 0, ErrHandlerTimeout
1221     }
1222     return tw.w.Write(p)
1223 }
1224
1225 func (tw *timeoutWriter) WriteHeader(code int) {
1226     tw.mu.Lock()
1227     if tw.timedOut || tw.wroteHeader {
1228         tw.mu.Unlock()

```

```
1229             return
1230         }
1231         tw.wroteHeader = true
1232         tw.mu.Unlock()
1233         tw.w.WriteHeader(code)
1234     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/sniff.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package http
6
7 import (
8     "bytes"
9     "encoding/binary"
10 )
11
12 // The algorithm uses at most sniffLen bytes to make its dec
13 const sniffLen = 512
14
15 // DetectContentType implements the algorithm described
16 // at http://mimesniff.spec.whatwg.org/ to determine the
17 // Content-Type of the given data. It considers at most the
18 // first 512 bytes of data. DetectContentType always return
19 // a valid MIME type: if it cannot determine a more specific
20 // returns "application/octet-stream".
21 func DetectContentType(data []byte) string {
22     if len(data) > sniffLen {
23         data = data[:sniffLen]
24     }
25
26     // Index of the first non-whitespace byte in data.
27     firstNonWS := 0
28     for ; firstNonWS < len(data) && isWS(data[firstNonWS]);
29     }
30
31     for _, sig := range sniffSignatures {
32         if ct := sig.match(data, firstNonWS); ct !=
33             return ct
34     }
35 }
36
37 return "application/octet-stream" // fallback
38 }
39
40 func isWS(b byte) bool {
41     return bytes.IndexByte([]byte("\t\n\x0C\r "), b) !=
42 }
43
44 type sniffSig interface {
```

```

45         // match returns the MIME type of the data, or "" if
46         match(data []byte, firstNonWS int) string
47     }
48
49     // Data matching the table in section 6.
50     var sniffSignatures = []sniffSig{
51         htmlSig("<!DOCTYPE HTML"),
52         htmlSig("<HTML"),
53         htmlSig("<HEAD"),
54         htmlSig("<SCRIPT"),
55         htmlSig("<IFRAME"),
56         htmlSig("<H1"),
57         htmlSig("<DIV"),
58         htmlSig("<FONT"),
59         htmlSig("<TABLE"),
60         htmlSig("<A"),
61         htmlSig("<STYLE"),
62         htmlSig("<TITLE"),
63         htmlSig("<B"),
64         htmlSig("<BODY"),
65         htmlSig("<BR"),
66         htmlSig("<P"),
67         htmlSig("<!--"),
68
69         &maskedSig{mask: []byte("\xFF\xFF\xFF\xFF\xFF"), pat
70
71         &exactSig{[]byte("%PDF-"), "application/pdf"},
72         &exactSig{[]byte("%!PS-Adobe-"), "application/postsc
73
74         // UTF BOMs.
75         &maskedSig{mask: []byte("\xFF\xFF\x00\x00"), pat: []
76         &maskedSig{mask: []byte("\xFF\xFF\x00\x00"), pat: []
77         &maskedSig{mask: []byte("\xFF\xFF\xFF\x00"), pat: []
78
79         &exactSig{[]byte("GIF87a"), "image/gif"},
80         &exactSig{[]byte("GIF89a"), "image/gif"},
81         &exactSig{[]byte("\x89\x50\x4E\x47\x0D\x0A\x1A\x0A")
82         &exactSig{[]byte("\xFF\xD8\xFF"), "image/jpeg"},
83         &exactSig{[]byte("BM"), "image/bmp"},
84         &maskedSig{
85             mask: []byte("\xFF\xFF\xFF\xFF\x00\x00\x00\x
86             pat: []byte("RIFF\x00\x00\x00\x00WEBPVP"),
87             ct: "image/webp",
88         },
89         &exactSig{[]byte("\x00\x00\x01\x00"), "image/vnd.mic
90         &exactSig{[]byte("\x4F\x67\x67\x53\x00"), "applicati
91         &maskedSig{
92             mask: []byte("\xFF\xFF\xFF\xFF\x00\x00\x00\x
93             pat: []byte("RIFF\x00\x00\x00\x00WAVE"),
94             ct: "audio/wave",

```

```

95     },
96     &exactSig{[]byte("\x1A\x45\xDF\xA3"), "video/webm"},
97     &exactSig{[]byte("\x52\x61\x72\x20\x1A\x07\x00"), "a
98     &exactSig{[]byte("\x50\x4B\x03\x04"), "application/z
99     &exactSig{[]byte("\x1F\x8B\x08"), "application/x-gzi
100
101     // TODO(dsymonds): Re-enable this when the spec is s
102     //mp4Sig(0),
103
104     textSig(0), // should be last
105 }
106
107 type exactSig struct {
108     sig []byte
109     ct  string
110 }
111
112 func (e *exactSig) match(data []byte, firstNonWS int) string
113     if bytes.HasPrefix(data, e.sig) {
114         return e.ct
115     }
116     return ""
117 }
118
119 type maskedSig struct {
120     mask, pat []byte
121     skipWS    bool
122     ct        string
123 }
124
125 func (m *maskedSig) match(data []byte, firstNonWS int) strin
126     if m.skipWS {
127         data = data[firstNonWS:]
128     }
129     if len(data) < len(m.mask) {
130         return ""
131     }
132     for i, mask := range m.mask {
133         db := data[i] & mask
134         if db != m.pat[i] {
135             return ""
136         }
137     }
138     return m.ct
139 }
140
141 type htmlSig []byte
142
143 func (h htmlSig) match(data []byte, firstNonWS int) string {

```

```

144     data = data[firstNonWS:]
145     if len(data) < len(h)+1 {
146         return ""
147     }
148     for i, b := range h {
149         db := data[i]
150         if 'A' <= b && b <= 'Z' {
151             db &= 0xDF
152         }
153         if b != db {
154             return ""
155         }
156     }
157     // Next byte must be space or right angle bracket.
158     if db := data[len(h)]; db != ' ' && db != '>' {
159         return ""
160     }
161     return "text/html; charset=utf-8"
162 }
163
164 type mp4Sig int
165
166 func (mp4Sig) match(data []byte, firstNonWS int) string {
167     // c.f. section 6.1.
168     if len(data) < 8 {
169         return ""
170     }
171     boxSize := int(binary.BigEndian.Uint32(data[:4]))
172     if boxSize%4 != 0 || len(data) < boxSize {
173         return ""
174     }
175     if !bytes.Equal(data[4:8], []byte("ftyp")) {
176         return ""
177     }
178     for st := 8; st < boxSize; st += 4 {
179         if st == 12 {
180             // minor version number
181             continue
182         }
183         seg := string(data[st : st+3])
184         switch seg {
185             case "mp4", "iso", "M4V", "M4P", "M4B":
186                 return "video/mp4"
187             /* The remainder are not in the spec
188             case "M4A":
189                 return "audio/mp4"
190             case "3gp":
191                 return "video/3gpp"
192             case "jp2":

```

```

193                                     return "image/jp2" // JPEG 2
194                                     */
195                                 }
196                             }
197                             return ""
198     }
199
200     type textSig int
201
202     func (textSig) match(data []byte, firstNonWS int) string {
203         // c.f. section 5, step 4.
204         for _, b := range data[firstNonWS:] {
205             switch {
206                 case 0x00 <= b && b <= 0x08,
207                     b == 0x0B,
208                     0x0E <= b && b <= 0x1A,
209                     0x1C <= b && b <= 0x1F:
210                 return ""
211             }
212         }
213         return "text/plain; charset=utf-8"
214     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/http/status.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package http
6
7 // HTTP status codes, defined in RFC 2616.
8 const (
9     StatusContinue           = 100
10    StatusSwitchingProtocols = 101
11
12    StatusOK                = 200
13    StatusCreated           = 201
14    StatusAccepted          = 202
15    StatusNonAuthoritativeInfo = 203
16    StatusNoContent         = 204
17    StatusResetContent      = 205
18    StatusPartialContent    = 206
19
20    StatusMultipleChoices    = 300
21    StatusMovedPermanently   = 301
22    StatusFound              = 302
23    StatusSeeOther           = 303
24    StatusNotModified        = 304
25    StatusUseProxy           = 305
26    StatusTemporaryRedirect  = 307
27
28    StatusBadRequest        = 400
29    StatusUnauthorized      = 401
30    StatusPaymentRequired   = 402
31    StatusForbidden         = 403
32    StatusNotFound          = 404
33    StatusMethodNotAllowed  = 405
34    StatusNotAcceptable     = 406
35    StatusProxyAuthRequired = 407
36    StatusRequestTimeout    = 408
37    StatusConflict          = 409
38    StatusGone              = 410
39    StatusLengthRequired    = 411
40    StatusPreconditionFailed = 412
41    StatusRequestEntityTooLarge = 413
42    StatusRequestURITooLong = 414
43    StatusUnsupportedMediaType = 415
44    StatusRequestedRangeNotSatisfiable = 416
```

```

45         StatusExpectationFailed           = 417
46         StatusTeapot                      = 418
47
48         StatusInternalServerError         = 500
49         StatusNotImplemented             = 501
50         StatusBadGateway                 = 502
51         StatusServiceUnavailable         = 503
52         StatusGatewayTimeout             = 504
53         StatusHTTPVersionNotSupported    = 505
54     )
55
56     var statusText = map[int]string{
57         StatusContinue:           "Continue",
58         StatusSwitchingProtocols: "Switching Protocols",
59
60         StatusOK:                  "OK",
61         StatusCreated:             "Created",
62         StatusAccepted:           "Accepted",
63         StatusNonAuthoritativeInfo: "Non-Authoritative Infor
64         StatusNoContent:          "No Content",
65         StatusResetContent:       "Reset Content",
66         StatusPartialContent:     "Partial Content",
67
68         StatusMultipleChoices:    "Multiple Choices",
69         StatusMovedPermanently:   "Moved Permanently",
70         StatusFound:              "Found",
71         StatusSeeOther:          "See Other",
72         StatusNotModified:       "Not Modified",
73         StatusUseProxy:          "Use Proxy",
74         StatusTemporaryRedirect:  "Temporary Redirect",
75
76         StatusBadRequest:         "Bad Request",
77         StatusUnauthorized:       "Unauthorized",
78         StatusPaymentRequired:    "Payment Require
79         StatusForbidden:         "Forbidden",
80         StatusNotFound:          "Not Found",
81         StatusMethodNotAllowed:   "Method Not Allo
82         StatusNotAcceptable:     "Not Acceptable"
83         StatusProxyAuthRequired:  "Proxy Authentic
84         StatusRequestTimeout:    "Request Timeout
85         StatusConflict:          "Conflict",
86         StatusGone:              "Gone",
87         StatusLengthRequired:    "Length Required
88         StatusPreconditionFailed: "Precondition Fa
89         StatusRequestEntityTooLarge: "Request Entity
90         StatusRequestURITooLong:  "Request URI Too
91         StatusUnsupportedMediaType: "Unsupported Med
92         StatusRequestedRangeNotSatisfiable: "Requested Range
93         StatusExpectationFailed:  "Expectation Fai
94         StatusTeapot:            "I'm a teapot",

```

```
95
96     StatusInternalServerError:    "Internal Server Error",
97     StatusNotImplemented:        "Not Implemented",
98     StatusBadGateway:            "Bad Gateway",
99     StatusServiceUnavailable:    "Service Unavailable"
100     StatusGatewayTimeout:        "Gateway Timeout",
101     StatusHTTPVersionNotSupported: "HTTP Version Not Supported"
102 }
103
104 // StatusText returns a text for the HTTP status code. It returns
105 // string if the code is unknown.
106 func StatusText(code int) string {
107     return statusText[code]
108 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/transfer.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package http
6
7 import (
8     "bufio"
9     "bytes"
10    "errors"
11    "fmt"
12    "io"
13    "io/ioutil"
14    "net/textproto"
15    "strconv"
16    "strings"
17 )
18
19 // transferWriter inspects the fields of a user-supplied Req
20 // sanitizes them without changing the user object and provi
21 // writing the respective header, body and trailer in wire f
22 type transferWriter struct {
23     Method          string
24     Body            io.Reader
25     BodyCloser     io.Closer
26     ResponseToHEAD bool
27     ContentLength  int64 // -1 means unknown, 0 means
28     Close          bool
29     TransferEncoding []string
30     Trailer       Header
31 }
32
33 func newTransferWriter(r interface{}) (t *transferWriter, er
34     t = &transferWriter{}
35
36     // Extract relevant fields
37     atLeastHTTP11 := false
38     switch rr := r.(type) {
39     case *Request:
40         if rr.ContentLength != 0 && rr.Body == nil {
41             return nil, fmt.Errorf("http: Reques
```

```

42     }
43     t.Method = rr.Method
44     t.Body = rr.Body
45     t.BodyCloser = rr.Body
46     t.ContentLength = rr.ContentLength
47     t.Close = rr.Close
48     t.TransferEncoding = rr.TransferEncoding
49     t.Trailer = rr.Trailer
50     atLeastHTTP11 = rr.ProtoAtLeast(1, 1)
51     if t.Body != nil && len(t.TransferEncoding)
52         if t.ContentLength == 0 {
53             // Test to see if it's actual
54             var buf [1]byte
55             n, _ := io.ReadFull(t.Body,
56                 if n == 1 {
57                     // Oh, guess there is
58                     // The ContentLength
59                     // Stich the Body back
60                     // consumed byte.
61                     t.ContentLength = -1
62                     t.Body = io.MultiReader
63                 } else {
64                     // Body is actually
65                     t.Body = nil
66                     t.BodyCloser = nil
67                 }
68             }
69             if t.ContentLength < 0 {
70                 t.TransferEncoding = []string
71             }
72         }
73     case *Response:
74         t.Method = rr.Request.Method
75         t.Body = rr.Body
76         t.BodyCloser = rr.Body
77         t.ContentLength = rr.ContentLength
78         t.Close = rr.Close
79         t.TransferEncoding = rr.TransferEncoding
80         t.Trailer = rr.Trailer
81         atLeastHTTP11 = rr.ProtoAtLeast(1, 1)
82         t.ResponseToHEAD = noBodyExpected(rr.Request)
83     }
84
85     // Sanitize Body, ContentLength, TransferEncoding
86     if t.ResponseToHEAD {
87         t.Body = nil
88         t.TransferEncoding = nil
89         // ContentLength is expected to hold Content
90         if t.ContentLength < 0 {
91             return nil, ErrMissingContentLength

```

```

92     }
93     } else {
94         if !atLeastHTTP11 || t.Body == nil {
95             t.TransferEncoding = nil
96         }
97         if chunked(t.TransferEncoding) {
98             t.ContentLength = -1
99         } else if t.Body == nil { // no chunking, no
100             t.ContentLength = 0
101         }
102     }
103
104     // Sanitize Trailer
105     if !chunked(t.TransferEncoding) {
106         t.Trailer = nil
107     }
108
109     return t, nil
110 }
111
112 func noBodyExpected(requestMethod string) bool {
113     return requestMethod == "HEAD"
114 }
115
116 func (t *transferWriter) shouldSendContentLength() bool {
117     if chunked(t.TransferEncoding) {
118         return false
119     }
120     if t.ContentLength > 0 {
121         return true
122     }
123     if t.ResponseToHEAD {
124         return true
125     }
126     // Many servers expect a Content-Length for these me
127     if t.Method == "POST" || t.Method == "PUT" {
128         return true
129     }
130     if t.ContentLength == 0 && isIdentity(t.TransferEnco
131         return true
132     }
133     return false
134 }
135
136
137 func (t *transferWriter) WriteHeader(w io.Writer) (err error
138     if t.Close {
139         _, err = io.WriteString(w, "Connection: clos
140         if err != nil {

```

```

141         return
142     }
143 }
144
145 // Write Content-Length and/or Transfer-Encoding who
146 // function of the sanitized field triple (Body, Con
147 // TransferEncoding)
148 if t.shouldSendContentLength() {
149     io.WriteString(w, "Content-Length: ")
150     _, err = io.WriteString(w, strconv.FormatInt
151     if err != nil {
152         return
153     }
154 } else if chunked(t.TransferEncoding) {
155     _, err = io.WriteString(w, "Transfer-Encodin
156     if err != nil {
157         return
158     }
159 }
160
161 // Write Trailer header
162 if t.Trailer != nil {
163     // TODO: At some point, there should be a ge
164     // writing long headers, using HTTP line spl
165     io.WriteString(w, "Trailer: ")
166     needComma := false
167     for k := range t.Trailer {
168         k = CanonicalHeaderKey(k)
169         switch k {
170             case "Transfer-Encoding", "Trailer",
171                 return &badStringError{"inva
172         }
173         if needComma {
174             io.WriteString(w, ",")
175         }
176         io.WriteString(w, k)
177         needComma = true
178     }
179     _, err = io.WriteString(w, "\r\n")
180 }
181
182 return
183 }
184
185 func (t *transferWriter) WriteBody(w io.Writer) (err error)
186     var ncopy int64
187
188     // Write body
189     if t.Body != nil {

```

```

190         if chunked(t.TransferEncoding) {
191             cw := newChunkedWriter(w)
192             _, err = io.Copy(cw, t.Body)
193             if err == nil {
194                 err = cw.Close()
195             }
196         } else if t.ContentLength == -1 {
197             ncopy, err = io.Copy(w, t.Body)
198         } else {
199             ncopy, err = io.Copy(w, io.LimitRead
200             nextra, err := io.Copy(ioutil.Discar
201             if err != nil {
202                 return err
203             }
204             ncopy += nextra
205         }
206         if err != nil {
207             return err
208         }
209         if err = t.BodyCloser.Close(); err != nil {
210             return err
211         }
212     }
213
214     if t.ContentLength != -1 && t.ContentLength != ncopy
215         return fmt.Errorf("http: Request.ContentLeng
216         t.ContentLength, ncopy)
217     }
218
219     // TODO(petar): Place trailer writer code here.
220     if chunked(t.TransferEncoding) {
221         // Last chunk, empty trailer
222         _, err = io.WriteString(w, "\r\n")
223     }
224
225     return
226 }
227
228 type transferReader struct {
229     // Input
230     Header      Header
231     StatusCode   int
232     RequestMethod string
233     ProtoMajor   int
234     ProtoMinor   int
235     // Output
236     Body          io.ReadCloser
237     ContentLength int64
238     TransferEncoding []string
239     Close          bool

```

```

240         Trailer          Header
241     }
242
243     // bodyAllowedForStatus returns whether a given response sta
244     // permits a body. See RFC2616, section 4.4.
245     func bodyAllowedForStatus(status int) bool {
246         switch {
247             case status >= 100 && status <= 199:
248                 return false
249             case status == 204:
250                 return false
251             case status == 304:
252                 return false
253         }
254         return true
255     }
256
257     // msg is *Request or *Response.
258     func readTransfer(msg interface{}, r *bufio.Reader) (err error) {
259         t := &transferReader{}
260
261         // Unify input
262         isResponse := false
263         switch rr := msg.(type) {
264             case *Response:
265                 t.Header = rr.Header
266                 t.StatusCode = rr.StatusCode
267                 t.RequestMethod = rr.Request.Method
268                 t.ProtoMajor = rr.ProtoMajor
269                 t.ProtoMinor = rr.ProtoMinor
270                 t.Close = shouldClose(t.ProtoMajor, t.ProtoM
271                 isResponse = true
272             case *Request:
273                 t.Header = rr.Header
274                 t.ProtoMajor = rr.ProtoMajor
275                 t.ProtoMinor = rr.ProtoMinor
276                 // Transfer semantics for Requests are exact
277                 // Responses with status code 200, respondin
278                 t.StatusCode = 200
279                 t.RequestMethod = "GET"
280             default:
281                 panic("unexpected type")
282         }
283
284         // Default to HTTP/1.1
285         if t.ProtoMajor == 0 && t.ProtoMinor == 0 {
286             t.ProtoMajor, t.ProtoMinor = 1, 1
287         }
288

```

```

289 // Transfer encoding, content length
290 t.TransferEncoding, err = fixTransferEncoding(t.Requ
291 if err != nil {
292     return err
293 }
294
295 t.ContentLength, err = fixLength(isResponse, t.Statu
296 if err != nil {
297     return err
298 }
299
300 // Trailer
301 t.Trailer, err = fixTrailer(t.Header, t.TransferEnco
302 if err != nil {
303     return err
304 }
305
306 // If there is no Content-Length or chunked Transfer
307 // and the status is not 1xx, 204 or 304, then the b
308 // See RFC2616, section 4.4.
309 switch msg.(type) {
310 case *Response:
311     if t.ContentLength == -1 &&
312         !chunked(t.TransferEncoding) &&
313         bodyAllowedForStatus(t.StatusCode) {
314         // Unbounded body.
315         t.Close = true
316     }
317 }
318
319 // Prepare body reader. ContentLength < 0 means chu
320 // or close connection when finished, since multipar
321 switch {
322 case chunked(t.TransferEncoding):
323     t.Body = &body{Reader: newChunkedReader(r),
324 case t.ContentLength >= 0:
325     // TODO: limit the Content-Length. This is a
326     t.Body = &body{Reader: io.LimitReader(r, t.C
327 default:
328     // t.ContentLength < 0, i.e. "Content-Length
329     if t.Close {
330         // Close semantics (i.e. HTTP/1.0)
331         t.Body = &body{Reader: r, closing: t
332     } else {
333         // Persistent connection (i.e. HTTP/
334         t.Body = &body{Reader: io.LimitReade
335     }
336 }
337

```

```

338 // Unify output
339 switch rr := msg.(type) {
340 case *Request:
341     rr.Body = t.Body
342     rr.ContentLength = t.ContentLength
343     rr.TransferEncoding = t.TransferEncoding
344     rr.Close = t.Close
345     rr.Trailer = t.Trailer
346 case *Response:
347     rr.Body = t.Body
348     rr.ContentLength = t.ContentLength
349     rr.TransferEncoding = t.TransferEncoding
350     rr.Close = t.Close
351     rr.Trailer = t.Trailer
352 }
353
354 return nil
355 }
356
357 // Checks whether chunked is part of the encodings stack
358 func chunked(te []string) bool { return len(te) > 0 && te[0]
359
360 // Checks whether the encoding is explicitly "identity".
361 func isIdentity(te []string) bool { return len(te) == 1 && t
362
363 // Sanitize transfer encoding
364 func fixTransferEncoding(requestMethod string, header Header
365     raw, present := header["Transfer-Encoding"]
366     if !present {
367         return nil, nil
368     }
369
370     delete(header, "Transfer-Encoding")
371
372     // Head responses have no bodies, so the transfer en
373     // should be ignored.
374     if requestMethod == "HEAD" {
375         return nil, nil
376     }
377
378     encodings := strings.Split(raw[0], ",")
379     te := make([]string, 0, len(encodings))
380     // TODO: Even though we only support "identity" and
381     // encodings, the loop below is designed with foresi
382     // invariant that must be maintained is that, if pre
383     // chunked encoding must always come first.
384     for _, encoding := range encodings {
385         encoding = strings.ToLower(strings.TrimSpace
386         // "identity" encoding is not recorded
387         if encoding == "identity" {

```

```

388             break
389         }
390         if encoding != "chunked" {
391             return nil, &badStringError{"unsuppo
392         }
393         te = te[0 : len(te)+1]
394         te[len(te)-1] = encoding
395     }
396     if len(te) > 1 {
397         return nil, &badStringError{"too many transf
398     }
399     if len(te) > 0 {
400         // Chunked encoding trumps Content-Length. S
401         // Section 4.4. Currently len(te) > 0 implic
402         // encoding.
403         delete(header, "Content-Length")
404         return te, nil
405     }
406     return nil, nil
407 }
408 }
409
410 // Determine the expected body length, using RFC 2616 Sectio
411 // function is not a method, because ultimately it should be
412 // ReadResponse and ReadRequest.
413 func fixLength(isResponse bool, status int, requestMethod st
414
415     // Logic based on response type or status
416     if noBodyExpected(requestMethod) {
417         return 0, nil
418     }
419     if status/100 == 1 {
420         return 0, nil
421     }
422     switch status {
423     case 204, 304:
424         return 0, nil
425     }
426
427     // Logic based on Transfer-Encoding
428     if chunked(te) {
429         return -1, nil
430     }
431
432     // Logic based on Content-Length
433     cl := strings.TrimSpace(header.Get("Content-Length"))
434     if cl != "" {
435         n, err := strconv.ParseInt(cl, 10, 64)
436         if err != nil || n < 0 {

```

```

437         return -1, &badStringError{"bad Cont
438     }
439     return n, nil
440 } else {
441     header.Del("Content-Length")
442 }
443
444 if !isResponse && requestMethod == "GET" {
445     // RFC 2616 doesn't explicitly permit nor fo
446     // entity-body on a GET request so we permit
447     // declared, but we default to 0 here (not -
448     // if there's no mention of a body.
449     return 0, nil
450 }
451
452 // Logic based on media type. The purpose of the fol
453 // to detect whether the unsupported "multipart/byte
454 // used. A proper Content-Type parser is needed in t
455 if strings.Contains(strings.ToLower(header.Get("Cont
456     return -1, ErrNotSupported
457 }
458
459 // Body-EOF logic based on other methods (like closi
460 return -1, nil
461 }
462
463 // Determine whether to hang up after sending a request and
464 // receiving a response and body
465 // 'header' is the request headers
466 func shouldClose(major, minor int, header Header) bool {
467     if major < 1 {
468         return true
469     } else if major == 1 && minor == 0 {
470         if !strings.Contains(strings.ToLower(header.
471             return true
472         }
473         return false
474     } else {
475         // TODO: Should split on commas, toss surrou
476         // and check each field.
477         if strings.ToLower(header.Get("Connection"))
478             header.Del("Connection")
479             return true
480     }
481 }
482 return false
483 }
484
485 // Parse the trailer header

```

```

486 func fixTrailer(header Header, te []string) (Header, error)
487     raw := header.Get("Trailer")
488     if raw == "" {
489         return nil, nil
490     }
491
492     header.Del("Trailer")
493     trailer := make(Header)
494     keys := strings.Split(raw, ",")
495     for _, key := range keys {
496         key = CanonicalHeaderKey(strings.TrimSpace(k
497         switch key {
498             case "Transfer-Encoding", "Trailer", "Conten
499                 return nil, &badStringError{"bad tra
500         }
501         trailer.Del(key)
502     }
503     if len(trailer) == 0 {
504         return nil, nil
505     }
506     if !chunked(te) {
507         // Trailer and no chunking
508         return nil, ErrUnexpectedTrailer
509     }
510     return trailer, nil
511 }
512
513 // body turns a Reader into a ReadCloser.
514 // Close ensures that the body has been fully read
515 // and then reads the trailer if necessary.
516 type body struct {
517     io.Reader
518     hdr      interface{} // non-nil (Response or Reques
519     r        *bufio.Reader // underlying wire-format read
520     closing bool        // is the connection to be clo
521     closed  bool
522
523     res *response // response writer for server requests
524 }
525
526 // ErrBodyReadAfterClose is returned when reading a Request
527 // the body has been closed. This typically happens when the
528 // read after an HTTP Handler calls WriteHeader or Write on
529 // ResponseWriter.
530 var ErrBodyReadAfterClose = errors.New("http: invalid Read o
531
532 func (b *body) Read(p []byte) (n int, err error) {
533     if b.closed {
534         return 0, ErrBodyReadAfterClose
535     }

```

```

536         n, err = b.Reader.Read(p)
537
538         // Read the final trailer once we hit EOF.
539         if err == io.EOF && b.hdr != nil {
540             if e := b.readTrailer(); e != nil {
541                 err = e
542             }
543             b.hdr = nil
544         }
545         return n, err
546     }
547
548     var (
549         singleCRLF = []byte("\r\n")
550         doubleCRLF = []byte("\r\n\r\n")
551     )
552
553     func seeUpcomingDoubleCRLF(r *bufio.Reader) bool {
554         for peekSize := 4; ; peekSize++ {
555             // This loop stops when Peek returns an error
556             // which it does when r's buffer has been filled
557             buf, err := r.Peek(peekSize)
558             if bytes.HasSuffix(buf, doubleCRLF) {
559                 return true
560             }
561             if err != nil {
562                 break
563             }
564         }
565         return false
566     }
567
568     func (b *body) readTrailer() error {
569         // The common case, since nobody uses trailers.
570         buf, _ := b.r.Peek(2)
571         if bytes.Equal(buf, singleCRLF) {
572             b.r.ReadByte()
573             b.r.ReadByte()
574             return nil
575         }
576
577         // Make sure there's a header terminator coming up,
578         // a DoS with an unbounded size Trailer. It's not a
579         // slip in a LimitReader here, as textproto.NewReader
580         // a concrete *bufio.Reader. Also, we can't get all
581         // back up to our conn's LimitedReader that *might*
582         // this bufio.Reader. Instead, a hack: we iterate
583         // to the bufio.Reader's max size, looking for a double
584         // This limits the trailer to the underlying buffer

```

```

585         if !seeUpcomingDoubleCRLF(b.r) {
586             return errors.New("http: suspiciously long t
587         }
588
589         hdr, err := textproto.NewReader(b.r).ReadMIMEHeader(
590         if err != nil {
591             return err
592         }
593         switch rr := b.hdr.(type) {
594         case *Request:
595             rr.Trailer = Header(hdr)
596         case *Response:
597             rr.Trailer = Header(hdr)
598         }
599         return nil
600     }
601
602     func (b *body) Close() error {
603         if b.closed {
604             return nil
605         }
606         defer func() {
607             b.closed = true
608         }()
609         if b.hdr == nil && b.closing {
610             // no trailer and closing the connection nex
611             // no point in reading to EOF.
612             return nil
613         }
614
615         // In a server request, don't continue reading from
616         // if we've already hit the maximum body size set by
617         // handler. If this is set, that also means the TCP
618         // is about to be closed, so getting to the next HTT
619         // in the stream is not necessary.
620         if b.res != nil && b.res.requestBodyLimitHit {
621             return nil
622         }
623
624         // Fully consume the body, which will also lead to u
625         // the trailer headers after the body, if present.
626         if _, err := io.Copy(ioutil.Discard, b); err != nil
627             return err
628         }
629         return nil
630     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/transport.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // HTTP client implementation. See RFC 2616.
6 //
7 // This is the low-level Transport implementation of RoundTr
8 // The high-level interface is in client.go.
9
10 package http
11
12 import (
13     "bufio"
14     "compress/gzip"
15     "crypto/tls"
16     "encoding/base64"
17     "errors"
18     "fmt"
19     "io"
20     "io/ioutil"
21     "log"
22     "net"
23     "net/url"
24     "os"
25     "strings"
26     "sync"
27 )
28
29 // DefaultTransport is the default implementation of Transpo
30 // used by DefaultClient. It establishes a new network conn
31 // each call to Do and uses HTTP proxies as directed by the
32 // $HTTP_PROXY and $NO_PROXY (or $http_proxy and $no_proxy)
33 // environment variables.
34 var DefaultTransport RoundTripper = &Transport{Proxy: ProxyF
35
36 // DefaultMaxIdleConnsPerHost is the default value of Transp
37 // MaxIdleConnsPerHost.
38 const DefaultMaxIdleConnsPerHost = 2
39
40 // Transport is an implementation of RoundTripper that suppo
41 // https, and http proxies (for either http or https with CC
```

```

42 // Transport can also cache connections for future re-use.
43 type Transport struct {
44     lk          sync.Mutex
45     idleConn map[string][]*persistConn
46     altProto map[string]RoundTripper // nil or map of UR
47
48     // TODO: tunable on global max cached connections
49     // TODO: tunable on timeout on cached connections
50     // TODO: optional pipelining
51
52     // Proxy specifies a function to return a proxy for
53     // Request. If the function returns a non-nil error,
54     // request is aborted with the provided error.
55     // If Proxy is nil or returns a nil *URL, no proxy is
56     Proxy func(*Request) (*url.URL, error)
57
58     // Dial specifies the dial function for creating TCP
59     // connections.
60     // If Dial is nil, net.Dial is used.
61     Dial func(net, addr string) (c net.Conn, err error)
62
63     // TLSClientConfig specifies the TLS configuration t
64     // tls.Client. If nil, the default configuration is
65     TLSClientConfig *tls.Config
66
67     DisableKeepAlives bool
68     DisableCompression bool
69
70     // MaxIdleConnsPerHost, if non-zero, controls the ma
71     // (keep-alive) to keep to keep per-host. If zero,
72     // DefaultMaxIdleConnsPerHost is used.
73     MaxIdleConnsPerHost int
74 }
75
76 // ProxyFromEnvironment returns the URL of the proxy to use
77 // given request, as indicated by the environment variables
78 // $HTTP_PROXY and $NO_PROXY (or $http_proxy and $no_proxy).
79 // An error is returned if the proxy environment is invalid.
80 // A nil URL and nil error are returned if no proxy is defin
81 // environment, or a proxy should not be used for the given
82 func ProxyFromEnvironment(req *Request) (*url.URL, error) {
83     proxy := getenvEitherCase("HTTP_PROXY")
84     if proxy == "" {
85         return nil, nil
86     }
87     if !useProxy(canonicalAddr(req.URL)) {
88         return nil, nil
89     }
90     proxyURL, err := url.Parse(proxy)
91     if err != nil || proxyURL.Scheme == "" {

```

```

92         if u, err := url.Parse("http://" + proxy); e
93             proxyURL = u
94             err = nil
95     }
96 }
97 if err != nil {
98     return nil, fmt.Errorf("invalid proxy address")
99 }
100 return proxyURL, nil
101 }
102
103 // ProxyURL returns a proxy function (for use in a Transport
104 // that always returns the same URL.
105 func ProxyURL(fixedURL *url.URL) func(*Request) (*url.URL, error) {
106     return func(*Request) (*url.URL, error) {
107         return fixedURL, nil
108     }
109 }
110
111 // transportRequest is a wrapper around a *Request that adds
112 // optional extra headers to write.
113 type transportRequest struct {
114     *Request // original request, not to be mutated
115     extra    Header // extra headers to write, or nil
116 }
117
118 func (tr *transportRequest) extraHeaders() Header {
119     if tr.extra == nil {
120         tr.extra = make(Header)
121     }
122     return tr.extra
123 }
124
125 // RoundTrip implements the RoundTripper interface.
126 func (t *Transport) RoundTrip(req *Request) (*Response, error) {
127     if req.URL == nil {
128         return nil, errors.New("http: nil Request.URL")
129     }
130     if req.Header == nil {
131         return nil, errors.New("http: nil Request.Header")
132     }
133     if req.URL.Scheme != "http" && req.URL.Scheme != "https" {
134         t.lk.Lock()
135         var rt RoundTripper
136         if t.altProto != nil {
137             rt = t.altProto[req.URL.Scheme]
138         }
139         t.lk.Unlock()
140         if rt == nil {

```

```

141         return nil, &badStringError{"unsuppo
142     }
143     return rt.RoundTrip(req)
144 }
145 treq := &transportRequest{Request: req}
146 cm, err := t.connectMethodForRequest(treq)
147 if err != nil {
148     return nil, err
149 }
150
151 // Get the cached or newly-created connection to eit
152 // host (for http or https), the http proxy, or the
153 // pre-CONNECTed to https server. In any case, we'l
154 // to send it requests.
155 pconn, err := t.getConn(cm)
156 if err != nil {
157     return nil, err
158 }
159
160 return pconn.roundTrip(treq)
161 }
162
163 // RegisterProtocol registers a new protocol with scheme.
164 // The Transport will pass requests using the given scheme t
165 // It is rt's responsibility to simulate HTTP request semant
166 //
167 // RegisterProtocol can be used by other packages to provide
168 // implementations of protocol schemes like "ftp" or "file".
169 func (t *Transport) RegisterProtocol(scheme string, rt Round
170     if scheme == "http" || scheme == "https" {
171         panic("protocol " + scheme + " already regis
172     }
173     t.lk.Lock()
174     defer t.lk.Unlock()
175     if t.altProto == nil {
176         t.altProto = make(map[string]RoundTripper)
177     }
178     if _, exists := t.altProto[scheme]; exists {
179         panic("protocol " + scheme + " already regis
180     }
181     t.altProto[scheme] = rt
182 }
183
184 // CloseIdleConnections closes any connections which were pr
185 // connected from previous requests but are now sitting idle
186 // a "keep-alive" state. It does not interrupt any connectio
187 // in use.
188 func (t *Transport) CloseIdleConnections() {
189     t.lk.Lock()

```

```

190         defer t.lk.Unlock()
191         if t.idleConn == nil {
192             return
193         }
194         for _, conns := range t.idleConn {
195             for _, pconn := range conns {
196                 pconn.close()
197             }
198         }
199         t.idleConn = make(map[string][]*persistConn)
200     }
201
202     //
203     // Private implementation past this point.
204     //
205
206     func getEnvEitherCase(k string) string {
207         if v := os.Getenv(strings.ToUpper(k)); v != "" {
208             return v
209         }
210         return os.Getenv(strings.ToLower(k))
211     }
212
213     func (t *Transport) connectMethodForRequest(treq *transportR
214         cm := &connectMethod{
215             targetScheme: treq.URL.Scheme,
216             targetAddr:    canonicalAddr(treq.URL),
217         }
218         if t.Proxy != nil {
219             var err error
220             cm.proxyURL, err = t.Proxy(treq.Request)
221             if err != nil {
222                 return nil, err
223             }
224         }
225         return cm, nil
226     }
227
228     // proxyAuth returns the Proxy-Authorization header to set
229     // on requests, if applicable.
230     func (cm *connectMethod) proxyAuth() string {
231         if cm.proxyURL == nil {
232             return ""
233         }
234         if u := cm.proxyURL.User; u != nil {
235             return "Basic " + base64.URLEncoding.EncodeT
236         }
237         return ""
238     }
239

```

```

240 // putIdleConn adds pconn to the list of idle persistent con
241 // a new request.
242 // If pconn is no longer needed or not in a good state, putI
243 // returns false.
244 func (t *Transport) putIdleConn(pconn *persistConn) bool {
245     t.lk.Lock()
246     defer t.lk.Unlock()
247     if t.DisableKeepAlives || t.MaxIdleConnsPerHost < 0
248         pconn.close()
249         return false
250     }
251     if pconn.isBroken() {
252         return false
253     }
254     key := pconn.cacheKey
255     max := t.MaxIdleConnsPerHost
256     if max == 0 {
257         max = DefaultMaxIdleConnsPerHost
258     }
259     if len(t.idleConn[key]) >= max {
260         pconn.close()
261         return false
262     }
263     t.idleConn[key] = append(t.idleConn[key], pconn)
264     return true
265 }
266
267 func (t *Transport) getIdleConn(cm *connectMethod) (pconn *p
268     t.lk.Lock()
269     defer t.lk.Unlock()
270     if t.idleConn == nil {
271         t.idleConn = make(map[string][]*persistConn)
272     }
273     key := cm.String()
274     for {
275         pconns, ok := t.idleConn[key]
276         if !ok {
277             return nil
278         }
279         if len(pconns) == 1 {
280             pconn = pconns[0]
281             delete(t.idleConn, key)
282         } else {
283             // 2 or more cached connections; pop
284             // TODO: queue?
285             pconn = pconns[len(pconns)-1]
286             t.idleConn[key] = pconns[0 : len(pco
287         }
288         if !pconn.isBroken() {

```

```

289             return
290         }
291     }
292     return
293 }
294
295 func (t *Transport) dial(network, addr string) (c net.Conn,
296     if t.Dial != nil {
297         return t.Dial(network, addr)
298     }
299     return net.Dial(network, addr)
300 }
301
302 // getConn dials and creates a new persistConn to the target
303 // specified in the connectMethod. This includes doing a pr
304 // and/or setting up TLS. If this doesn't return an error,
305 // is ready to write requests to.
306 func (t *Transport) getConn(cm *connectMethod) (*persistConn
307     if pc := t.getIdleConn(cm); pc != nil {
308         return pc, nil
309     }
310
311     conn, err := t.dial("tcp", cm.addr())
312     if err != nil {
313         if cm.proxyURL != nil {
314             err = fmt.Errorf("http: error connec
315         }
316         return nil, err
317     }
318
319     pa := cm.proxyAuth()
320
321     pconn := &persistConn{
322         t:          t,
323         cacheKey: cm.String(),
324         conn:       conn,
325         reqch:      make(chan requestAndChan, 50),
326     }
327
328     switch {
329     case cm.proxyURL == nil:
330         // Do nothing.
331     case cm.targetScheme == "http":
332         pconn.isProxy = true
333         if pa != "" {
334             pconn.mutateHeaderFunc = func(h Head
335                 h.Set("Proxy-Authorization",
336             }
337     }

```

```

338     case cm.targetScheme == "https":
339         connectReq := &Request{
340             Method: "CONNECT",
341             URL:    &url.URL{Opaque: cm.targetAd
342             Host:   cm.targetAddr,
343             Header: make(Header),
344         }
345         if pa != "" {
346             connectReq.Header.Set("Proxy-Authori
347         }
348         connectReq.Write(conn)
349
350         // Read response.
351         // Okay to use and discard buffered reader h
352         // TLS server will not speak until spoken to
353         br := bufio.NewReader(conn)
354         resp, err := ReadResponse(br, connectReq)
355         if err != nil {
356             conn.Close()
357             return nil, err
358         }
359         if resp.StatusCode != 200 {
360             f := strings.SplitN(resp.Status, " "
361             conn.Close()
362             return nil, errors.New(f[1])
363         }
364     }
365
366     if cm.targetScheme == "https" {
367         // Initiate TLS and check remote host name a
368         conn = tls.Client(conn, t.TLSClientConfig)
369         if err = conn.(*tls.Conn).Handshake(); err !
370             return nil, err
371     }
372     if t.TLSClientConfig == nil || !t.TLSClientC
373         if err = conn.(*tls.Conn).VerifyHost
374             return nil, err
375     }
376 }
377 pconn.conn = conn
378 }
379
380 pconn.br = bufio.NewReader(pconn.conn)
381 pconn.bw = bufio.NewWriter(pconn.conn)
382 go pconn.readLoop()
383 return pconn, nil
384 }
385
386 // useProxy returns true if requests to addr should use a pr
387 // according to the NO_PROXY or no_proxy environment variabl

```

```

388 // addr is always a canonicalAddr with a host and port.
389 func useProxy(addr string) bool {
390     if len(addr) == 0 {
391         return true
392     }
393     host, _, err := net.SplitHostPort(addr)
394     if err != nil {
395         return false
396     }
397     if host == "localhost" {
398         return false
399     }
400     if ip := net.ParseIP(host); ip != nil {
401         if ip.IsLoopback() {
402             return false
403         }
404     }
405
406     no_proxy := getenvEitherCase("NO_PROXY")
407     if no_proxy == "*" {
408         return false
409     }
410
411     addr = strings.ToLower(strings.TrimSpace(addr))
412     if hasPort(addr) {
413         addr = addr[:strings.LastIndex(addr, ":")]
414     }
415
416     for _, p := range strings.Split(no_proxy, ",") {
417         p = strings.ToLower(strings.TrimSpace(p))
418         if len(p) == 0 {
419             continue
420         }
421         if hasPort(p) {
422             p = p[:strings.LastIndex(p, ":")]
423         }
424         if addr == p || (p[0] == '.' && (strings.Has
425             return false
426         }
427     }
428     return true
429 }
430
431 // connectMethod is the map key (in its String form) for kee
432 // TCP connections alive for subsequent HTTP requests.
433 //
434 // A connect method may be of the following types:
435 //
436 // Cache key form          Description

```

```

437 // -----
438 // ||http|foo.com          http directly to server, no
439 // ||https|foo.com        https directly to server, n
440 // http://proxy.com|https|foo.com http to proxy, then CONNE
441 // http://proxy.com|http          http to proxy, http to an
442 //
443 // Note: no support to https to the proxy yet.
444 //
445 type connectMethod struct {
446     proxyURL      *url.URL // nil for no proxy, else full
447     targetScheme string // "http" or "https"
448     targetAddr   string // Not used if proxy + http ta
449 }
450
451 func (ck *connectMethod) String() string {
452     proxyStr := ""
453     if ck.proxyURL != nil {
454         proxyStr = ck.proxyURL.String()
455     }
456     return strings.Join([]string{proxyStr, ck.targetSche
457 }
458
459 // addr returns the first hop "host:port" to which we need t
460 func (cm *connectMethod) addr() string {
461     if cm.proxyURL != nil {
462         return canonicalAddr(cm.proxyURL)
463     }
464     return cm.targetAddr
465 }
466
467 // tlsHost returns the host name to match against the peer's
468 // TLS certificate.
469 func (cm *connectMethod) tlsHost() string {
470     h := cm.targetAddr
471     if hasPort(h) {
472         h = h[:strings.LastIndex(h, ":")]
473     }
474     return h
475 }
476
477 // persistConn wraps a connection, usually a persistent one
478 // (but may be used for non-keep-alive requests as well)
479 type persistConn struct {
480     t      *Transport
481     cacheKey string // its connectMethod.String()
482     conn   net.Conn
483     br     *bufio.Reader // from conn
484     bw     *bufio.Writer // to conn
485     reqch  chan requestAndChan // written by roundTrip

```

```

486         isProxy    bool
487
488         // mutateHeaderFunc is an optional func to modify ex
489         // headers on each outbound request before it's writ
490         // original Request given to RoundTrip is not modifi
491         mutateHeaderFunc func(Header)
492
493         lk          sync.Mutex // guards numExpecte
494         numExpectedResponses int
495         broken      bool // an error has happened o
496     }
497
498     func (pc *persistConn) isBroken() bool {
499         pc.lk.Lock()
500         defer pc.lk.Unlock()
501         return pc.broken
502     }
503
504     var remoteSideClosedFunc func(error) bool // or nil to use d
505
506     func remoteSideClosed(err error) bool {
507         if err == io.EOF {
508             return true
509         }
510         if remoteSideClosedFunc != nil {
511             return remoteSideClosedFunc(err)
512         }
513         return false
514     }
515
516     func (pc *persistConn) readLoop() {
517         alive := true
518         var lastbody io.ReadCloser // last response body, if
519
520         for alive {
521             pb, err := pc.br.Peek(1)
522
523             pc.lk.Lock()
524             if pc.numExpectedResponses == 0 {
525                 pc.closeLocked()
526                 pc.lk.Unlock()
527                 if len(pb) > 0 {
528                     log.Printf("Unsolicited resp
529                               string(pb), err)
530                 }
531                 return
532             }
533             pc.lk.Unlock()
534
535             rc := <-pc.reqch

```

```

536
537 // Advance past the previous response's body
538 // caller hasn't done so.
539 if lastbody != nil {
540     lastbody.Close() // assumed idempote
541     lastbody = nil
542 }
543 resp, err := ReadResponse(pc.br, rc.req)
544
545 if err != nil {
546     pc.close()
547 } else {
548     hasBody := rc.req.Method != "HEAD" &
549     if rc.addedGzip && hasBody && resp.H
550     resp.Header.Del("Content-Enc
551     resp.Header.Del("Content-Len
552     resp.ContentLength = -1
553     gzReader, zerr := gzip.NewRe
554     if zerr != nil {
555         pc.close()
556         err = zerr
557     } else {
558         resp.Body = &readFir
559     }
560     }
561     resp.Body = &bodyEOFSignal{body: res
562 }
563
564 if err != nil || resp.Close || rc.req.Close
565     alive = false
566 }
567
568 hasBody := resp != nil && resp.ContentLength
569 var waitForBodyRead chan bool
570 if alive {
571     if hasBody {
572         lastbody = resp.Body
573         waitForBodyRead = make(chan
574         resp.Body.(*bodyEOFSignal).f
575         if !pc.t.putIdleConn
576             alive = fals
577         }
578         waitForBodyRead <- t
579     }
580 } else {
581     // When there's no response
582     // reuse the TCP connection
583     // we need to prevent Client
584     // closing the Response.Body

```

```

585         // loop, otherwise it might
586         // before the client code ha
587         // read it (even though it'l
588         lastbody = nil
589
590         if !pc.t.putIdleConn(pc) {
591             alive = false
592         }
593     }
594 }
595
596 rc.ch <- responseAndError{resp, err}
597
598     // Wait for the just-returned response body
599     // before we race and peek on the underlying
600     if waitForBodyRead != nil {
601         <-waitForBodyRead
602     }
603 }
604 }
605
606 type responseAndError struct {
607     res *Response
608     err error
609 }
610
611 type requestAndChan struct {
612     req *Request
613     ch  chan responseAndError
614
615     // did the Transport (as opposed to the client code)
616     // Accept-Encoding gzip header? only if it we set it
617     // we transparently decode the gzip.
618     addedGzip bool
619 }
620
621 func (pc *persistConn) roundTrip(req *transportRequest) (res
622     if pc.mutateHeaderFunc != nil {
623         pc.mutateHeaderFunc(req.extraHeaders())
624     }
625
626     // Ask for a compressed version if the caller didn't
627     // own value for Accept-Encoding. We only attempted
628     // uncompress the gzip stream if we were the layer t
629     // requested it.
630     requestedGzip := false
631     if !pc.t.DisableCompression && req.Header.Get("Accep
632         // Request gzip only, not deflate. Deflate i
633         // not as universally supported anyway.

```

```

634             // See: http://www.gzip.org/zlib/zlib\_faq.ht
635             requestedGzip = true
636             req.extraHeaders().Set("Accept-Encoding", "g
637         }
638
639         pc.lk.Lock()
640         pc.numExpectedResponses++
641         pc.lk.Unlock()
642
643         err = req.Request.write(pc.bw, pc.isProxy, req.extra
644         if err != nil {
645             pc.close()
646             return
647         }
648         pc.bw.Flush()
649
650         ch := make(chan responseAndError, 1)
651         pc.reqch <- requestAndChan{req.Request, ch, requeste
652         re := <-ch
653         pc.lk.Lock()
654         pc.numExpectedResponses--
655         pc.lk.Unlock()
656
657         return re.res, re.err
658     }
659
660     func (pc *persistConn) close() {
661         pc.lk.Lock()
662         defer pc.lk.Unlock()
663         pc.closeLocked()
664     }
665
666     func (pc *persistConn) closeLocked() {
667         pc.broken = true
668         pc.conn.Close()
669         pc.mutateHeaderFunc = nil
670     }
671
672     var portMap = map[string]string{
673         "http": "80",
674         "https": "443",
675     }
676
677     // canonicalAddr returns url.Host but always with a ":port"
678     func canonicalAddr(url *url.URL) string {
679         addr := url.Host
680         if !hasPort(addr) {
681             return addr + ":" + portMap[url.Scheme]
682         }
683         return addr

```

```

684 }
685
686 func responseIsKeepAlive(res *Response) bool {
687     // TODO: implement. for now just always shutting do
688     return false
689 }
690
691 // bodyEOFSignal wraps a ReadCloser but runs fn (if non-nil)
692 // once, right before the final Read() or Close() call retur
693 // EOF has been seen.
694 type bodyEOFSignal struct {
695     body      io.ReadCloser
696     fn        func()
697     isClosed  bool
698 }
699
700 func (es *bodyEOFSignal) Read(p []byte) (n int, err error) {
701     n, err = es.body.Read(p)
702     if es.isClosed && n > 0 {
703         panic("http: unexpected bodyEOFSignal Read a
704     }
705     if err == io.EOF && es.fn != nil {
706         es.fn()
707         es.fn = nil
708     }
709     return
710 }
711
712 func (es *bodyEOFSignal) Close() (err error) {
713     if es.isClosed {
714         return nil
715     }
716     es.isClosed = true
717     err = es.body.Close()
718     if err == nil && es.fn != nil {
719         es.fn()
720         es.fn = nil
721     }
722     return
723 }
724
725 type readFirstCloseBoth struct {
726     io.ReadCloser
727     io.Closer
728 }
729
730 func (r *readFirstCloseBoth) Close() error {
731     if err := r.ReadCloser.Close(); err != nil {
732         r.Closer.Close()

```

```
733         return err
734     }
735     if err := r.Closer.Close(); err != nil {
736         return err
737     }
738     return nil
739 }
740
741 // discardOnCloseReadCloser consumes all its input on Close.
742 type discardOnCloseReadCloser struct {
743     io.ReadCloser
744 }
745
746 func (d *discardOnCloseReadCloser) Close() error {
747     io.Copy(ioutil.Discard, d.ReadCloser) // ignore error
748     return d.ReadCloser.Close()
749 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/cgi/child.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file implements CGI from the perspective of a child
6 // process.
7
8 package cgi
9
10 import (
11     "bufio"
12     "crypto/tls"
13     "errors"
14     "fmt"
15     "io"
16     "io/ioutil"
17     "net"
18     "net/http"
19     "net/url"
20     "os"
21     "strconv"
22     "strings"
23 )
24
25 // Request returns the HTTP request as represented in the cu
26 // environment. This assumes the current program is being ru
27 // by a web server in a CGI environment.
28 // The returned Request's Body is populated, if applicable.
29 func Request() (*http.Request, error) {
30     r, err := RequestFromMap(envMap(os.Environ()))
31     if err != nil {
32         return nil, err
33     }
34     if r.ContentLength > 0 {
35         r.Body = ioutil.NopCloser(io.LimitReader(os.
36     })
37     return r, nil
38 }
39
40 func envMap(env []string) map[string]string {
41     m := make(map[string]string)
```

```

42     for _, kv := range env {
43         if idx := strings.Index(kv, "="); idx != -1
44             m[kv[:idx]] = kv[idx+1:]
45     }
46 }
47 return m
48 }
49
50 // RequestFromMap creates an http.Request from CGI variables
51 // The returned Request's Body field is not populated.
52 func RequestFromMap(params map[string]string) (*http.Request
53     r := new(http.Request)
54     r.Method = params["REQUEST_METHOD"]
55     if r.Method == "" {
56         return nil, errors.New("cgi: no REQUEST_METH
57     }
58
59     r.Proto = params["SERVER_PROTOCOL"]
60     var ok bool
61     r.ProtoMajor, r.ProtoMinor, ok = http.ParseHTTPVersi
62     if !ok {
63         return nil, errors.New("cgi: invalid SERVER_
64     }
65
66     r.Close = true
67     r.Trailer = http.Header{}
68     r.Header = http.Header{}
69
70     r.Host = params["HTTP_HOST"]
71
72     if lenstr := params["CONTENT_LENGTH"]; lenstr != ""
73         clen, err := strconv.ParseInt(lenstr, 10, 64
74         if err != nil {
75             return nil, errors.New("cgi: bad CON
76         }
77         r.ContentLength = clen
78     }
79
80     if ct := params["CONTENT_TYPE"]; ct != "" {
81         r.Header.Set("Content-Type", ct)
82     }
83
84     // Copy "HTTP_FOO_BAR" variables to "Foo-Bar" Header
85     for k, v := range params {
86         if !strings.HasPrefix(k, "HTTP_") || k == "H
87             continue
88         }
89         r.Header.Add(strings.Replace(k[5:], "_", "-")
90     }
91

```

```

92         // TODO: cookies.  parsing them isn't exported, thou
93
94         if r.Host != "" {
95             // Hostname is provided, so we can reasonabl
96             // even if we have to assume 'http' for the
97             rawurl := "http://" + r.Host + params["REQUE
98             url, err := url.Parse(rawurl)
99             if err != nil {
100                 return nil, errors.New("cgi: failed
101             }
102             r.URL = url
103         }
104         // Fallback logic if we don't have a Host header or
105         // failed to parse
106         if r.URL == nil {
107             uriStr := params["REQUEST_URI"]
108             url, err := url.Parse(uriStr)
109             if err != nil {
110                 return nil, errors.New("cgi: failed
111             }
112             r.URL = url
113         }
114
115         // There's apparently a de-facto standard for this.
116         // http://docstore.mik.ua/orelly/linux/cgi/ch03_02.h
117         if s := params["HTTPS"]; s == "on" || s == "ON" || s
118             r.TLS = &tls.ConnectionState{HandshakeComple
119         }
120
121         // Request.RemoteAddr has its port set by Go's stand
122         // server, so we do here too. We don't have one, tho
123         // use a dummy one.
124         r.RemoteAddr = net.JoinHostPort(params["REMOTE_ADDR"
125
126         return r, nil
127     }
128
129     // Serve executes the provided Handler on the currently acti
130     // request, if any. If there's no current CGI environment
131     // an error is returned. The provided handler may be nil to
132     // http.DefaultServeMux.
133     func Serve(handler http.Handler) error {
134         req, err := Request()
135         if err != nil {
136             return err
137         }
138         if handler == nil {
139             handler = http.DefaultServeMux
140         }

```

```

141     rw := &response{
142         req:    req,
143         header: make(http.Header),
144         bufw:   bufio.NewWriter(os.Stdout),
145     }
146     handler.ServeHTTP(rw, req)
147     rw.Write(nil) // make sure a response is sent
148     if err = rw.bufw.Flush(); err != nil {
149         return err
150     }
151     return nil
152 }
153
154 type response struct {
155     req        *http.Request
156     header     http.Header
157     bufw       *bufio.Writer
158     headerSent bool
159 }
160
161 func (r *response) Flush() {
162     r.bufw.Flush()
163 }
164
165 func (r *response) Header() http.Header {
166     return r.header
167 }
168
169 func (r *response) Write(p []byte) (n int, err error) {
170     if !r.headerSent {
171         r.WriteHeader(http.StatusOK)
172     }
173     return r.bufw.Write(p)
174 }
175
176 func (r *response) WriteHeader(code int) {
177     if r.headerSent {
178         // Note: explicitly using Stderr, as Stdout
179         fmt.Fprintf(os.Stderr, "CGI attempted to wri
180         return
181     }
182     r.headerSent = true
183     fmt.Fprintf(r.bufw, "Status: %d %s\r\n", code, http.
184
185     // Set a default Content-Type
186     if _, hasType := r.header["Content-Type"]; !hasType
187         r.header.Add("Content-Type", "text/html; cha
188     }
189

```

```
190         r.header.Write(r.bufw)
191         r.bufw.WriteString("\r\n")
192         r.bufw.Flush()
193     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/cgi/host.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file implements the host side of CGI (being the webs
6 // parent process).
7
8 // Package cgi implements CGI (Common Gateway Interface) as
9 // in RFC 3875.
10 //
11 // Note that using CGI means starting a new process to handl
12 // request, which is typically less efficient than using a
13 // long-running server. This package is intended primarily
14 // compatibility with existing systems.
15 package cgi
16
17 import (
18     "bufio"
19     "fmt"
20     "io"
21     "log"
22     "net/http"
23     "os"
24     "os/exec"
25     "path/filepath"
26     "regexp"
27     "runtime"
28     "strconv"
29     "strings"
30 )
31
32 var trailingPort = regexp.MustCompile(`:([0-9]+)$`)
33
34 var osDefaultInheritEnv = map[string][]string{
35     "darwin": {"DYLD_LIBRARY_PATH"},
36     "freebsd": {"LD_LIBRARY_PATH"},
37     "hpux": {"LD_LIBRARY_PATH", "SHLIB_PATH"},
38     "irix": {"LD_LIBRARY_PATH", "LD_LIBRARYN32_PATH"},
39     "linux": {"LD_LIBRARY_PATH"},
40     "openbsd": {"LD_LIBRARY_PATH"},
41     "solaris": {"LD_LIBRARY_PATH", "LD_LIBRARY_PATH_32",
```

```

42         "windows": {"SystemRoot", "COMSPEC", "PATHEXT", "WIN
43     }
44
45 // Handler runs an executable in a subprocess with a CGI env
46 type Handler struct {
47     Path string // path to the CGI executable
48     Root string // root URI prefix of handler or empty f
49
50     // Dir specifies the CGI executable's working direct
51     // If Dir is empty, the base directory of Path is us
52     // If Path has no base directory, the current workin
53     // directory is used.
54     Dir string
55
56     Env []string // extra environment variable
57     InheritEnv []string // environment variables to i
58     Logger *log.Logger // optional log for errors or
59     Args []string // optional arguments to pass
60
61     // PathLocationHandler specifies the root http Handl
62     // should handle internal redirects when the CGI pro
63     // returns a Location header value starting with a "
64     // specified in RFC 3875 § 6.3.2. This will likely b
65     // http.DefaultServeMux.
66     //
67     // If nil, a CGI response with a local URI path is i
68     // back to the client and not redirected internally.
69     PathLocationHandler http.Handler
70 }
71
72 // removeLeadingDuplicates remove leading duplicate in envir
73 // It's possible to override environment like following.
74 //     cgi.Handler{
75 //         ...
76 //         Env: []string{"SCRIPT_FILENAME=foo.php"},
77 //     }
78 func removeLeadingDuplicates(env []string) (ret []string) {
79     n := len(env)
80     for i := 0; i < n; i++ {
81         e := env[i]
82         s := strings.SplitN(e, "=", 2)[0]
83         found := false
84         for j := i + 1; j < n; j++ {
85             if s == strings.SplitN(env[j], "=",
86                 found = true
87                 break
88             }
89         }
90         if !found {
91             ret = append(ret, e)

```

```

92         }
93     }
94     return
95 }
96
97 func (h *Handler) ServeHTTP(rw http.ResponseWriter, req *http.Request) {
98     root := h.Root
99     if root == "" {
100         root = "/"
101     }
102
103     if len(req.TransferEncoding) > 0 && req.TransferEncoding[0] != "chunked" {
104         rw.WriteHeader(http.StatusBadRequest)
105         rw.Write([]byte("Chunked request bodies are not supported"))
106         return
107     }
108
109     pathInfo := req.URL.Path
110     if root != "/" && strings.HasPrefix(pathInfo, root) {
111         pathInfo = pathInfo[len(root):]
112     }
113
114     port := "80"
115     if matches := trailingPort.FindStringSubmatch(req.Host); len(matches) > 0 {
116         port = matches[1]
117     }
118
119     env := []string{
120         "SERVER_SOFTWARE=go",
121         "SERVER_NAME=" + req.Host,
122         "SERVER_PROTOCOL=HTTP/1.1",
123         "HTTP_HOST=" + req.Host,
124         "GATEWAY_INTERFACE=CGI/1.1",
125         "REQUEST_METHOD=" + req.Method,
126         "QUERY_STRING=" + req.URL.RawQuery,
127         "REQUEST_URI=" + req.URL.RequestURI(),
128         "PATH_INFO=" + pathInfo,
129         "SCRIPT_NAME=" + root,
130         "SCRIPT_FILENAME=" + h.Path,
131         "REMOTE_ADDR=" + req.RemoteAddr,
132         "REMOTE_HOST=" + req.RemoteAddr,
133         "SERVER_PORT=" + port,
134     }
135
136     if req.TLS != nil {
137         env = append(env, "HTTPS=on")
138     }
139
140     for k, v := range req.Header {

```

```

141         k = strings.Map(upperCaseAndUnderscore, k)
142         joinStr := ", "
143         if k == "COOKIE" {
144             joinStr = "; "
145         }
146         env = append(env, "HTTP_"+k+"="+strings.Join
147     }
148
149     if req.ContentLength > 0 {
150         env = append(env, fmt.Sprintf("CONTENT_LENGTH="))
151     }
152     if ctype := req.Header.Get("Content-Type"); ctype != "" {
153         env = append(env, "CONTENT_TYPE="+ctype)
154     }
155
156     if h.Env != nil {
157         env = append(env, h.Env...)
158     }
159
160     envPath := os.Getenv("PATH")
161     if envPath == "" {
162         envPath = "/bin:/usr/bin:/usr/ucb:/usr/bsd:/usr/local/bin"
163     }
164     env = append(env, "PATH="+envPath)
165
166     for _, e := range h.InheritEnv {
167         if v := os.Getenv(e); v != "" {
168             env = append(env, e+"="+v)
169         }
170     }
171
172     for _, e := range osDefaultInheritEnv[runtime.GOOS] {
173         if v := os.Getenv(e); v != "" {
174             env = append(env, e+"="+v)
175         }
176     }
177
178     env = removeLeadingDuplicates(env)
179
180     var cwd, path string
181     if h.Dir != "" {
182         path = h.Path
183         cwd = h.Dir
184     } else {
185         cwd, path = filepath.Split(h.Path)
186     }
187     if cwd == "" {
188         cwd = "."
189     }

```

```

190
191     internalError := func(err error) {
192         rw.WriteHeader(http.StatusInternalServerError)
193         h.printf("CGI error: %v", err)
194     }
195
196     cmd := &exec.Cmd{
197         Path:    path,
198         Args:    append([]string{h.Path}, h.Args...),
199         Dir:     cwd,
200         Env:     env,
201         Stderr:  os.Stderr, // for now
202     }
203     if req.ContentLength != 0 {
204         cmd.Stdin = req.Body
205     }
206     stdoutRead, err := cmd.StdoutPipe()
207     if err != nil {
208         internalError(err)
209         return
210     }
211
212     err = cmd.Start()
213     if err != nil {
214         internalError(err)
215         return
216     }
217     defer cmd.Wait()
218     defer stdoutRead.Close()
219
220     linebody := bufio.NewReaderSize(stdoutRead, 1024)
221     headers := make(http.Header)
222     statusCode := 0
223     for {
224         line, isPrefix, err := linebody.ReadLine()
225         if isPrefix {
226             rw.WriteHeader(http.StatusInternalServerError)
227             h.printf("cgi: long header line from")
228             return
229         }
230         if err == io.EOF {
231             break
232         }
233         if err != nil {
234             rw.WriteHeader(http.StatusInternalServerError)
235             h.printf("cgi: error reading headers")
236             return
237         }
238         if len(line) == 0 {
239             break

```

```

240     }
241     parts := strings.SplitN(string(line), ":", 2)
242     if len(parts) < 2 {
243         h.printf("cgi: bogus header line: %s\n", line)
244         continue
245     }
246     header, val := parts[0], parts[1]
247     header = strings.TrimSpace(header)
248     val = strings.TrimSpace(val)
249     switch {
250     case header == "Status":
251         if len(val) < 3 {
252             h.printf("cgi: bogus status\n", line)
253             return
254         }
255         code, err := strconv.Atoi(val[0:3])
256         if err != nil {
257             h.printf("cgi: bogus status: %s\n", line)
258             h.printf("cgi: line was %q", line)
259             return
260         }
261         statusCode = code
262     default:
263         headers.Add(header, val)
264     }
265 }
266
267 if loc := headers.Get("Location"); loc != "" {
268     if strings.HasPrefix(loc, "/") && h.PathLocal {
269         h.handleInternalRedirect(rw, req, loc)
270         return
271     }
272     if statusCode == 0 {
273         statusCode = http.StatusFound
274     }
275 }
276
277 if statusCode == 0 {
278     statusCode = http.StatusOK
279 }
280
281 // Copy headers to rw's headers, after we've decided
282 // go into handleInternalRedirect, which won't want
283 // headers to have been touched.
284 for k, vv := range headers {
285     for _, v := range vv {
286         rw.Header().Add(k, v)
287     }
288 }

```

```

289
290     rw.WriteHeader(statusCode)
291
292     _, err = io.Copy(rw, linebody)
293     if err != nil {
294         h.printf("cgi: copy error: %v", err)
295     }
296 }
297
298 func (h *Handler) printf(format string, v ...interface{}) {
299     if h.Logger != nil {
300         h.Logger.Printf(format, v...)
301     } else {
302         log.Printf(format, v...)
303     }
304 }
305
306 func (h *Handler) handleInternalRedirect(rw http.ResponseWriter
307     url, err := req.URL.Parse(path)
308     if err != nil {
309         rw.WriteHeader(http.StatusInternalServerError)
310         h.printf("cgi: error resolving local URI pat
311         return
312     }
313     // TODO: RFC 3875 isn't clear if only GET is support
314     // suggests so: "Note that any message-body attached
315     // request (such as for a POST request) may not be a
316     // to the resource that is the target of the redirec
317     // should do some tests against Apache to see how it
318     // POST, HEAD, etc. Does the internal redirect get t
319     // method or just GET? What about incoming headers?
320     // (e.g. Cookies) Which headers, if any, are copied
321     // second request?
322     newReq := &http.Request{
323         Method:     "GET",
324         URL:        url,
325         Proto:      "HTTP/1.1",
326         ProtoMajor: 1,
327         ProtoMinor: 1,
328         Header:     make(http.Header),
329         Host:       url.Host,
330         RemoteAddr: req.RemoteAddr,
331         TLS:        req.TLS,
332     }
333     h.PathLocationHandler.ServeHTTP(rw, newReq)
334 }
335
336 func upperCaseAndUnderscore(r rune) rune {
337     switch {

```

```
338     case r >= 'a' && r <= 'z':
339         return r - ('a' - 'A')
340     case r == '-':
341         return '_'
342     case r == '=':
343         // Maybe not part of the CGI 'spec' but woul
344         // the environment in any case, as Go repres
345         // environment as a slice of "key=value" str
346         return '_'
347     }
348     // TODO: other transformations in spec or practice?
349     return r
350 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/cgi/child.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package cgi
6
7 // This file implements FastCGI from the perspective of a ch
8
9 import (
10     "errors"
11     "fmt"
12     "io"
13     "net"
14     "net/http"
15     "net/http/cgi"
16     "os"
17     "time"
18 )
19
20 // request holds the state for an in-progress request. As so
21 // it's converted to an http.Request.
22 type request struct {
23     pw          *io.PipeWriter
24     reqId       uint16
25     params      map[string]string
26     buf         [1024]byte
27     rawParams   []byte
28     keepConn    bool
29 }
30
31 func newRequest(reqId uint16, flags uint8) *request {
32     r := &request{
33         reqId:       reqId,
34         params:      map[string]string{},
35         keepConn:    flags&flagKeepConn != 0,
36     }
37     r.rawParams = r.buf[:0]
38     return r
39 }
40
41 // parseParams reads an encoded []byte into Params.
```

```

42 func (r *request) parseParams() {
43     text := r.rawParams
44     r.rawParams = nil
45     for len(text) > 0 {
46         keyLen, n := readSize(text)
47         if n == 0 {
48             return
49         }
50         text = text[n:]
51         valLen, n := readSize(text)
52         if n == 0 {
53             return
54         }
55         text = text[n:]
56         key := readString(text, keyLen)
57         text = text[keyLen:]
58         val := readString(text, valLen)
59         text = text[valLen:]
60         r.params[key] = val
61     }
62 }
63
64 // response implements http.ResponseWriter.
65 type response struct {
66     req      *request
67     header   http.Header
68     w        *bufio.Writer
69     wroteHeader bool
70 }
71
72 func newResponse(c *child, req *request) *response {
73     return &response{
74         req:      req,
75         header:   http.Header{},
76         w:        newWriter(c.conn, typeStdout, req.re
77     }
78 }
79
80 func (r *response) Header() http.Header {
81     return r.header
82 }
83
84 func (r *response) Write(data []byte) (int, error) {
85     if !r.wroteHeader {
86         r.WriteHeader(http.StatusOK)
87     }
88     return r.w.Write(data)
89 }
90
91 func (r *response) WriteHeader(code int) {

```

```

92     if r.wroteHeader {
93         return
94     }
95     r.wroteHeader = true
96     if code == http.StatusNotModified {
97         // Must not have body.
98         r.header.Del("Content-Type")
99         r.header.Del("Content-Length")
100        r.header.Del("Transfer-Encoding")
101    } else if r.header.Get("Content-Type") == "" {
102        r.header.Set("Content-Type", "text/html; cha
103    }
104
105    if r.header.Get("Date") == "" {
106        r.header.Set("Date", time.Now().UTC().Format
107    }
108
109    fmt.Fprintf(r.w, "Status: %d %s\r\n", code, http.Sta
110    r.header.Write(r.w)
111    r.w.WriteString("\r\n")
112 }
113
114 func (r *response) Flush() {
115     if !r.wroteHeader {
116         r.WriteHeader(http.StatusOK)
117     }
118     r.w.Flush()
119 }
120
121 func (r *response) Close() error {
122     r.Flush()
123     return r.w.Close()
124 }
125
126 type child struct {
127     conn      *conn
128     handler   http.Handler
129     requests  map[uint16]*request // keyed by request ID
130 }
131
132 func newChild(rwc io.ReadWriteCloser, handler http.Handler)
133     return &child{
134         conn:      newConn(rwc),
135         handler:   handler,
136         requests:  make(map[uint16]*request),
137     }
138 }
139
140 func (c *child) serve() {

```

```

141     defer c.conn.Close()
142     var rec record
143     for {
144         if err := rec.read(c.conn.rwc); err != nil {
145             return
146         }
147         if err := c.handleRecord(&rec); err != nil {
148             return
149         }
150     }
151 }
152
153 var errCloseConn = errors.New("fcgi: connection should be cl
154
155 func (c *child) handleRecord(rec *record) error {
156     req, ok := c.requests[rec.h.Id]
157     if !ok && rec.h.Type != typeBeginRequest && rec.h.Ty
158         // The spec says to ignore unknown request I
159         return nil
160     }
161     if ok && rec.h.Type == typeBeginRequest {
162         // The server is trying to begin a request w
163         // as an in-progress request. This is an err
164         return errors.New("fcgi: received ID that is
165     }
166
167     switch rec.h.Type {
168     case typeBeginRequest:
169         var br beginRequest
170         if err := br.read(rec.content()); err != nil
171             return err
172         }
173         if br.role != roleResponder {
174             c.conn.writeEndRequest(rec.h.Id, 0,
175                 return nil
176         }
177         c.requests[rec.h.Id] = newRequest(rec.h.Id,
178     case typeParams:
179         // NOTE(eds): Technically a key-value pair c
180         // between two packets. We buffer until we've
181         if len(rec.content()) > 0 {
182             req.rawParams = append(req.rawParams
183                 return nil
184         }
185         req.parseParams()
186     case typeStdin:
187         content := rec.content()
188         if req.pw == nil {
189             var body io.ReadCloser

```

```

190         if len(content) > 0 {
191             // body could be an io.Limit
192             // as long as both sides are
193             // body, req.pw = io.Pipe()
194         }
195         go c.serveRequest(req, body)
196     }
197     if len(content) > 0 {
198         // TODO(eds): This blocks until the
199         // If the handler takes a long time,
200         req.pw.Write(content)
201     } else if req.pw != nil {
202         req.pw.Close()
203     }
204     case typeGetValues:
205         values := map[string]string{"FCGI_MPXS_CONNS"}
206         c.conn.writePairs(typeGetValuesResult, 0, va
207     case typeData:
208         // If the filter role is implemented, read t
209     case typeAbortRequest:
210         delete(c.requests, rec.h.Id)
211         c.conn.writeEndRequest(rec.h.Id, 0, statusRe
212         if !req.keepConn {
213             // connection will close upon return
214             return errCloseConn
215         }
216     default:
217         b := make([]byte, 8)
218         b[0] = byte(rec.h.Type)
219         c.conn.writeRecord(typeUnknownType, 0, b)
220     }
221     return nil
222 }
223
224 func (c *child) serveRequest(req *request, body io.ReadClose
225     r := newResponse(c, req)
226     httpReq, err := cgi.RequestFromMap(req.params)
227     if err != nil {
228         // there was an error reading the request
229         r.WriteHeader(http.StatusInternalServerError)
230         c.conn.writeRecord(typeStderr, req.reqId, []
231     } else {
232         httpReq.Body = body
233         c.handler.ServeHTTP(r, httpReq)
234     }
235     if body != nil {
236         body.Close()
237     }
238     r.Close()
239     c.conn.writeEndRequest(req.reqId, 0, statusRequestCo

```

```

240         if !req.keepConn {
241             c.conn.Close()
242         }
243     }
244
245     // Serve accepts incoming FastCGI connections on the listene
246     // goroutine for each. The goroutine reads requests and then
247     // to reply to them.
248     // If l is nil, Serve accepts connections from os.Stdin.
249     // If handler is nil, http.DefaultServeMux is used.
250     func Serve(l net.Listener, handler http.Handler) error {
251         if l == nil {
252             var err error
253             l, err = net.FileListener(os.Stdin)
254             if err != nil {
255                 return err
256             }
257             defer l.Close()
258         }
259         if handler == nil {
260             handler = http.DefaultServeMux
261         }
262         for {
263             rw, err := l.Accept()
264             if err != nil {
265                 return err
266             }
267             c := newChild(rw, handler)
268             go c.serve()
269         }
270         panic("unreachable")
271     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/cgi/cgi.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package cgi implements the FastCGI protocol.
6 // Currently only the responder role is supported.
7 // The protocol is defined at http://www.fastcgi.com/drupal/
8 package cgi
9
10 // This file defines the raw protocol and some utilities use
11 // the host.
12
13 import (
14     "bufio"
15     "bytes"
16     "encoding/binary"
17     "errors"
18     "io"
19     "sync"
20 )
21
22 // recType is a record type, as defined by
23 // http://www.fastcgi.com/devkit/doc/cgi-spec.html#S8
24 type recType uint8
25
26 const (
27     typeBeginRequest      recType = 1
28     typeAbortRequest      recType = 2
29     typeEndRequest        recType = 3
30     typeParams            recType = 4
31     typeStdin             recType = 5
32     typeStdout            recType = 6
33     typeStderr            recType = 7
34     typeData              recType = 8
35     typeGetValues         recType = 9
36     typeGetValuesResult   recType = 10
37     typeUnknownType      recType = 11
38 )
39
40 // keep the connection between web-server and responder open
41 const flagKeepConn = 1
```

```

42
43 const (
44     maxWrite = 65535 // maximum record body
45     maxPad   = 255
46 )
47
48 const (
49     roleResponder = iota + 1 // only Responders are impl
50     roleAuthorizer
51     roleFilter
52 )
53
54 const (
55     statusRequestComplete = iota
56     statusCantMultiplex
57     statusOverloaded
58     statusUnknownRole
59 )
60
61 const headerLen = 8
62
63 type header struct {
64     Version      uint8
65     Type         recType
66     Id           uint16
67     ContentLength uint16
68     PaddingLength uint8
69     Reserved     uint8
70 }
71
72 type beginRequest struct {
73     role      uint16
74     flags     uint8
75     reserved [5]uint8
76 }
77
78 func (br *beginRequest) read(content []byte) error {
79     if len(content) != 8 {
80         return errors.New("fcgi: invalid begin reque
81     }
82     br.role = binary.BigEndian.Uint16(content)
83     br.flags = content[2]
84     return nil
85 }
86
87 // for padding so we don't have to allocate all the time
88 // not synchronized because we don't care what the contents
89 var pad [maxPad]byte
90
91 func (h *header) init(recType recType, reqId uint16, content

```

```

92         h.Version = 1
93         h.Type = recType
94         h.Id = reqId
95         h.ContentLength = uint16(contentLength)
96         h.PaddingLength = uint8(-contentLength & 7)
97     }
98
99     // conn sends records over rwc
100    type conn struct {
101        mutex sync.Mutex
102        rwc   io.ReadWriteCloser
103
104        // to avoid allocations
105        buf bytes.Buffer
106        h   header
107    }
108
109    func newConn(rwc io.ReadWriteCloser) *conn {
110        return &conn{rwc: rwc}
111    }
112
113    func (c *conn) Close() error {
114        c.mutex.Lock()
115        defer c.mutex.Unlock()
116        return c.rwc.Close()
117    }
118
119    type record struct {
120        h   header
121        buf [maxWrite + maxPad]byte
122    }
123
124    func (rec *record) read(r io.Reader) (err error) {
125        if err = binary.Read(r, binary.BigEndian, &rec.h); err != nil {
126            return err
127        }
128        if rec.h.Version != 1 {
129            return errors.New("fcgi: invalid header vers")
130        }
131        n := int(rec.h.ContentLength) + int(rec.h.PaddingLength)
132        if _, err = io.ReadFull(r, rec.buf[:n]); err != nil {
133            return err
134        }
135        return nil
136    }
137
138    func (r *record) content() []byte {
139        return r.buf[:r.h.ContentLength]
140    }

```

```

141
142 // writeRecord writes and sends a single record.
143 func (c *conn) writeRecord(recType recType, reqId uint16, b
144     c.mutex.Lock()
145     defer c.mutex.Unlock()
146     c.buf.Reset()
147     c.h.init(recType, reqId, len(b))
148     if err := binary.Write(&c.buf, binary.BigEndian, c.h
149         return err
150     }
151     if _, err := c.buf.Write(b); err != nil {
152         return err
153     }
154     if _, err := c.buf.Write(pad[:c.h.PaddingLength]); e
155         return err
156     }
157     _, err := c.rwc.Write(c.buf.Bytes())
158     return err
159 }
160
161 func (c *conn) writeBeginRequest(reqId uint16, role uint16,
162     b := [8]byte{byte(role >> 8), byte(role), flags}
163     return c.writeRecord(typeBeginRequest, reqId, b[:])
164 }
165
166 func (c *conn) writeEndRequest(reqId uint16, appStatus int,
167     b := make([]byte, 8)
168     binary.BigEndian.PutUint32(b, uint32(appStatus))
169     b[4] = protocolStatus
170     return c.writeRecord(typeEndRequest, reqId, b)
171 }
172
173 func (c *conn) writePairs(recType recType, reqId uint16, pai
174     w := newWriter(c, recType, reqId)
175     b := make([]byte, 8)
176     for k, v := range pairs {
177         n := encodeSize(b, uint32(len(k)))
178         n += encodeSize(b[n:], uint32(len(v)))
179         if _, err := w.Write(b[:n]); err != nil {
180             return err
181         }
182         if _, err := w.WriteString(k); err != nil {
183             return err
184         }
185         if _, err := w.WriteString(v); err != nil {
186             return err
187         }
188     }
189     w.Close()

```

```

190         return nil
191     }
192
193     func readSize(s []byte) (uint32, int) {
194         if len(s) == 0 {
195             return 0, 0
196         }
197         size, n := uint32(s[0]), 1
198         if size & (1 << 7) != 0 {
199             if len(s) < 4 {
200                 return 0, 0
201             }
202             n = 4
203             size = binary.BigEndian.Uint32(s)
204             size &^= 1 << 31
205         }
206         return size, n
207     }
208
209     func readString(s []byte, size uint32) string {
210         if size > uint32(len(s)) {
211             return ""
212         }
213         return string(s[:size])
214     }
215
216     func encodeSize(b []byte, size uint32) int {
217         if size > 127 {
218             size |= 1 << 31
219             binary.BigEndian.PutUint32(b, size)
220             return 4
221         }
222         b[0] = byte(size)
223         return 1
224     }
225
226     // bufWriter encapsulates bufio.Writer but also closes the u
227     // Closed.
228     type bufWriter struct {
229         closer io.Closer
230         *bufio.Writer
231     }
232
233     func (w *bufWriter) Close() error {
234         if err := w.Writer.Flush(); err != nil {
235             w.closer.Close()
236             return err
237         }
238         return w.closer.Close()
239     }

```

```

240
241 func newWriter(c *conn, recType recType, reqId uint16) *bufw
242     s := &streamWriter{c: c, recType: recType, reqId: re
243     w := bufio.NewWriterSize(s, maxWrite)
244     return &bufWriter{s, w}
245 }
246
247 // streamWriter abstracts out the separation of a stream int
248 // It only writes maxWrite bytes at a time.
249 type streamWriter struct {
250     c      *conn
251     recType recType
252     reqId  uint16
253 }
254
255 func (w *streamWriter) Write(p []byte) (int, error) {
256     nn := 0
257     for len(p) > 0 {
258         n := len(p)
259         if n > maxWrite {
260             n = maxWrite
261         }
262         if err := w.c.writeRecord(w.recType, w.reqId
263             return nn, err
264         }
265         nn += n
266         p = p[n:]
267     }
268     return nn, nil
269 }
270
271 func (w *streamWriter) Close() error {
272     // send empty record to close the stream
273     return w.c.writeRecord(w.recType, w.reqId, nil)
274 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/httpptest/recorder.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package httpptest provides utilities for HTTP testing.
6 package httpptest
7
8 import (
9     "bytes"
10    "net/http"
11 )
12
13 // ResponseRecorder is an implementation of http.ResponseWriter
14 // records its mutations for later inspection in tests.
15 type ResponseRecorder struct {
16     Code      int // the HTTP response code fr
17     HeaderMap http.Header // the HTTP response headers
18     Body      *bytes.Buffer // if non-nil, the bytes.Buf
19     Flushed   bool
20 }
21
22 // NewRecorder returns an initialized ResponseRecorder.
23 func NewRecorder() *ResponseRecorder {
24     return &ResponseRecorder{
25         HeaderMap: make(http.Header),
26         Body:      new(bytes.Buffer),
27     }
28 }
29
30 // DefaultRemoteAddr is the default remote address to return
31 // an explicit DefaultRemoteAddr isn't set on ResponseRecord
32 const DefaultRemoteAddr = "1.2.3.4"
33
34 // Header returns the response headers.
35 func (rw *ResponseRecorder) Header() http.Header {
36     return rw.HeaderMap
37 }
38
39 // Write always succeeds and writes to rw.Body, if not nil.
40 func (rw *ResponseRecorder) Write(buf []byte) (int, error) {
41     if rw.Body != nil {
```

```
42         rw.Body.Write(buf)
43     }
44     if rw.Code == 0 {
45         rw.Code = http.StatusOK
46     }
47     return len(buf), nil
48 }
49
50 // WriteHeader sets rw.Code.
51 func (rw *ResponseRecorder) WriteHeader(code int) {
52     rw.Code = code
53 }
54
55 // Flush sets rw.Flushed to true.
56 func (rw *ResponseRecorder) Flush() {
57     rw.Flushed = true
58 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/httpptest/server.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Implementation of Server
6
7 package httpptest
8
9 import (
10     "crypto/tls"
11     "flag"
12     "fmt"
13     "net"
14     "net/http"
15     "os"
16     "sync"
17 )
18
19 // A Server is an HTTP server listening on a system-chosen p
20 // local loopback interface, for use in end-to-end HTTP test
21 type Server struct {
22     URL      string // base URL of form http://ipaddr:po
23     Listener net.Listener
24     TLS      *tls.Config // nil if not using using TLS
25
26     // Config may be changed after calling NewUnstartedS
27     // before Start or StartTLS.
28     Config *http.Server
29
30     // wg counts the number of outstanding HTTP requests
31     // Close blocks until all requests are finished.
32     wg sync.WaitGroup
33 }
34
35 // historyListener keeps track of all connections that it's
36 // accepted.
37 type historyListener struct {
38     net.Listener
39     history []net.Conn
40 }
41
```

```

42 func (hs *historyListener) Accept() (c net.Conn, err error)
43     c, err = hs.Listener.Accept()
44     if err == nil {
45         hs.history = append(hs.history, c)
46     }
47     return
48 }
49
50 func newLocalListener() net.Listener {
51     if *serve != "" {
52         l, err := net.Listen("tcp", *serve)
53         if err != nil {
54             panic(fmt.Sprintf("httptest: failed
55         })
56         return l
57     }
58     l, err := net.Listen("tcp", "127.0.0.1:0")
59     if err != nil {
60         if l, err = net.Listen("tcp6", "[::1]:0"); e
61             panic(fmt.Sprintf("httptest: failed
62         })
63     }
64     return l
65 }
66
67 // When debugging a particular http server-based test,
68 // this flag lets you run
69 //     go test -run=BrokenTest -httptest.serve=127.0.0.1:80
70 // to start the broken server so you can interact with it ma
71 var serve = flag.String("httptest.serve", "", "if non-empty,
72
73 // NewServer starts and returns a new Server.
74 // The caller should call Close when finished, to shut it do
75 func NewServer(handler http.Handler) *Server {
76     ts := NewUnstartedServer(handler)
77     ts.Start()
78     return ts
79 }
80
81 // NewUnstartedServer returns a new Server but doesn't start
82 //
83 // After changing its configuration, the caller should call
84 // StartTLS.
85 //
86 // The caller should call Close when finished, to shut it do
87 func NewUnstartedServer(handler http.Handler) *Server {
88     return &Server{
89         Listener: newLocalListener(),
90         Config:   &http.Server{Handler: handler},
91     }

```

```

92 }
93
94 // Start starts a server from NewUnstartedServer.
95 func (s *Server) Start() {
96     if s.URL != "" {
97         panic("Server already started")
98     }
99     s.Listener = &historyListener{s.Listener, make([]net
100 s.URL = "http://" + s.Listener.Addr().String()
101 s.wrapHandler()
102 go s.Config.Serve(s.Listener)
103 if *serve != "" {
104     fmt.Fprintln(os.Stderr, "httptest: serving o
105     select {}
106 }
107 }
108
109 // StartTLS starts TLS on a server from NewUnstartedServer.
110 func (s *Server) StartTLS() {
111     if s.URL != "" {
112         panic("Server already started")
113     }
114     cert, err := tls.X509KeyPair(localhostCert, localhos
115     if err != nil {
116         panic(fmt.Sprintf("httptest: NewTLSServer: %
117     }
118
119     s.TLS = &tls.Config{
120         NextProtos: []string{"http/1.1"},
121         Certificates: []tls.Certificate{cert},
122     }
123     tlsListener := tls.NewListener(s.Listener, s.TLS)
124
125     s.Listener = &historyListener{tlsListener, make([]ne
126     s.URL = "https://" + s.Listener.Addr().String()
127     s.wrapHandler()
128     go s.Config.Serve(s.Listener)
129 }
130
131 func (s *Server) wrapHandler() {
132     h := s.Config.Handler
133     if h == nil {
134         h = http.DefaultServeMux
135     }
136     s.Config.Handler = &waitGroupHandler{
137         s: s,
138         h: h,
139     }
140 }

```

```

141
142 // NewTLSserver starts and returns a new Server using TLS.
143 // The caller should call Close when finished, to shut it do
144 func NewTLSserver(handler http.Handler) *Server {
145     ts := NewUnstartedServer(handler)
146     ts.StartTLS()
147     return ts
148 }
149
150 // Close shuts down the server and blocks until all outstand
151 // requests on this server have completed.
152 func (s *Server) Close() {
153     s.Listener.Close()
154     s.wg.Wait()
155 }
156
157 // CloseClientConnections closes any currently open HTTP con
158 // to the test Server.
159 func (s *Server) CloseClientConnections() {
160     hl, ok := s.Listener.(*historyListener)
161     if !ok {
162         return
163     }
164     for _, conn := range hl.history {
165         conn.Close()
166     }
167 }
168
169 // waitGroupHandler wraps a handler, incrementing and decrem
170 // sync.WaitGroup on each request, to enable Server.Close to
171 // until outstanding requests are finished.
172 type waitGroupHandler struct {
173     s *Server
174     h http.Handler // non-nil
175 }
176
177 func (h *waitGroupHandler) ServeHTTP(w http.ResponseWriter,
178     h.s.wg.Add(1)
179     defer h.s.wg.Done() // a defer, in case ServeHTTP be
180     h.h.ServeHTTP(w, r)
181 }
182
183 // localhostCert is a PEM-encoded TLS cert with SAN DNS name
184 // "127.0.0.1" and "[::1]", expiring at the last second of 2
185 // of ASN.1 time).
186 var localhostCert = []byte(`-----BEGIN CERTIFICATE-----
187 MIIBOTCB5qADAgEAgEAMAsGCSqGSIB3DQEBAAMB4XDTcwMDEwMTAwMDAw
188 DTQ5MTIzMTIzNTk1OVowADBAMAsGCSqGSIB3DQEBAQNLADBIAKeAsuA5mAFM
189 qoBzcvKzIq4kzuT5epSp2AkcQfyBHm7K13Ws7u+0b5Vb9gqTf5cAiIKcrtrX

```

```
190 8i1UQF6AzwIDAQABo08wTTA0BgNVHQ8BAf8EBAMCACQwDQYDVR00BAYEBAEC
191 DwYDVR0jBAGwBoAEAQIDBDAbBgNVHREEFDASggkxMjcuMC4wLjGCBVs60jFd
192 CSqGSIb3DQEBBQNBAJH30zjLWRztrWpOCgJL8RQWLaKzhK79pVhAx6q/3NrF
193 +l1BRZstTwIGdoGId8BRpErK1TXkniFb95ZMynM=
194 -----END CERTIFICATE-----
195 `)
196
197 // localhostKey is the private key for localhostCert.
198 var localhostKey = []byte(`-----BEGIN RSA PRIVATE KEY-----
199 MIIBPQIBAAJBALLgOZgBTI+k06qAc3LysyKuJM7k+XqUqdgJHEH8gR5uytd1
200 tG+VW/YKk3+XAIiCnK7a11apC/ItVEBegM8CAwEAAQJBAI5sxq7naeR9ahyq
201 SIv2iMxLuPEHaezf5CY0PwjSjBPyVhyRevkhtqEjF/wkgL7C2nWpYHsUCBDE
202 3KECIQDtEGB2uInkZAahl3WuJziXGLB+p8Wgx7wzSM6bHu1c6QIhAMEp++Ca
203 /TrU0zwY/fw4SvQeb49BPZUF3oqR8Xz3AiEA1rAJHBzBgd0QKdE3ksMUPcnc
204 poCcELmz2clVXtkCIQCLytuLV38XHToTipR4yMl60+6arZajZ56uq7m7ZRV0
205 AM65XA0w8Dsg9Kq78aYXi0EDc5DL0sbFUu/S1mRcCg93
206 -----END RSA PRIVATE KEY-----
207 `)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/httputil/chunked.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // The wire protocol for HTTP's "chunked" Transfer-Encoding.
6
7 // This code is a duplicate of ../chunked.go with these edit
8 //     s/newChunked/NewChunked/g
9 //     s/package http/package httputil/
10 // Please make any changes in both files.
11
12 package httputil
13
14 import (
15     "bufio"
16     "bytes"
17     "errors"
18     "io"
19     "strconv"
20 )
21
22 const maxLineLength = 4096 // assumed <= bufio.defaultBufSiz
23
24 var ErrLineTooLong = errors.New("header line too long")
25
26 // NewChunkedReader returns a new chunkedReader that transla
27 // out of HTTP "chunked" format before returning it.
28 // The chunkedReader returns io.EOF when the final 0-length
29 //
30 // NewChunkedReader is not needed by normal applications. Th
31 // automatically decodes chunking when reading response bodi
32 func NewChunkedReader(r io.Reader) io.Reader {
33     br, ok := r.(*bufio.Reader)
34     if !ok {
35         br = bufio.NewReader(r)
36     }
37     return &chunkedReader{r: br}
38 }
39
40 type chunkedReader struct {
41     r *bufio.Reader
```

```

42         n    uint64 // unread bytes in chunk
43         err error
44     }
45
46     func (cr *chunkedReader) beginChunk() {
47         // chunk-size CRLF
48         var line string
49         line, cr.err = readLine(cr.r)
50         if cr.err != nil {
51             return
52         }
53         cr.n, cr.err = strconv.ParseUint(line, 16, 64)
54         if cr.err != nil {
55             return
56         }
57         if cr.n == 0 {
58             cr.err = io.EOF
59         }
60     }
61
62     func (cr *chunkedReader) Read(b []uint8) (n int, err error) {
63         if cr.err != nil {
64             return 0, cr.err
65         }
66         if cr.n == 0 {
67             cr.beginChunk()
68             if cr.err != nil {
69                 return 0, cr.err
70             }
71         }
72         if uint64(len(b)) > cr.n {
73             b = b[0:cr.n]
74         }
75         n, cr.err = cr.r.Read(b)
76         cr.n -= uint64(n)
77         if cr.n == 0 && cr.err == nil {
78             // end of chunk (CRLF)
79             b := make([]byte, 2)
80             if _, cr.err = io.ReadFull(cr.r, b); cr.err == nil {
81                 if b[0] != '\r' || b[1] != '\n' {
82                     cr.err = errors.New("malformed chunk")
83                 }
84             }
85         }
86         return n, cr.err
87     }
88
89     // Read a line of bytes (up to \n) from b.
90     // Give up if the line exceeds maxLineLength.
91     // The returned bytes are a pointer into storage in

```

```

92 // the bufio, so they are only valid until the next bufio re
93 func readLineBytes(b *bufio.Reader) (p []byte, err error) {
94     if p, err = b.ReadSlice('\n'); err != nil {
95         // We always know when EOF is coming.
96         // If the caller asked for a line, there sho
97         if err == io.EOF {
98             err = io.ErrUnexpectedEOF
99         } else if err == bufio.ErrBufferFull {
100             err = ErrLineTooLong
101         }
102         return nil, err
103     }
104     if len(p) >= maxLineLength {
105         return nil, ErrLineTooLong
106     }
107
108     // Chop off trailing white space.
109     p = bytes.TrimRight(p, " \r\t\n")
110
111     return p, nil
112 }
113
114 // readLineBytes, but convert the bytes into a string.
115 func readLine(b *bufio.Reader) (s string, err error) {
116     p, e := readLineBytes(b)
117     if e != nil {
118         return "", e
119     }
120     return string(p), nil
121 }
122
123 // NewChunkedWriter returns a new chunkedWriter that transla
124 // "chunked" format before writing them to w. Closing the re
125 // sends the final 0-length chunk that marks the end of the
126 //
127 // NewChunkedWriter is not needed by normal applications. Th
128 // package adds chunking automatically if handlers don't set
129 // Content-Length header. Using NewChunkedWriter inside a ha
130 // would result in double chunking or chunking with a Conten
131 // length, both of which are wrong.
132 func NewChunkedWriter(w io.Writer) io.WriteCloser {
133     return &chunkedWriter{w}
134 }
135
136 // Writing to chunkedWriter translates to writing in HTTP ch
137 // Encoding wire format to the underlying Wire chunkedWriter
138 type chunkedWriter struct {
139     Wire io.Writer
140 }

```

```

141
142 // Write the contents of data as one chunk to Wire.
143 // NOTE: Note that the corresponding chunk-writing procedure
144 // a bug since it does not check for success of io.WriteStri
145 func (cw *chunkedWriter) Write(data []byte) (n int, err erro
146
147         // Don't send 0-length data. It looks like EOF for c
148         if len(data) == 0 {
149             return 0, nil
150         }
151
152         head := strconv.FormatInt(int64(len(data)), 16) + "\
153
154         if _, err = io.WriteString(cw.Wire, head); err != ni
155             return 0, err
156         }
157         if n, err = cw.Wire.Write(data); err != nil {
158             return
159         }
160         if n != len(data) {
161             err = io.ErrShortWrite
162             return
163         }
164         _, err = io.WriteString(cw.Wire, "\r\n")
165
166         return
167     }
168
169     func (cw *chunkedWriter) Close() error {
170         _, err := io.WriteString(cw.Wire, "0\r\n")
171         return err
172     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/httputil/dump.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package httputil
6
7 import (
8     "bufio"
9     "bytes"
10    "fmt"
11    "io"
12    "io/ioutil"
13    "net"
14    "net/http"
15    "net/url"
16    "strings"
17    "time"
18 )
19
20 // One of the copies, say from b to r2, could be avoided by
21 // elaborate trick where the other copy is made during Reque
22 // This would complicate things too much, given that these f
23 // debugging only.
24 func drainBody(b io.ReadCloser) (r1, r2 io.ReadCloser, err e
25     var buf bytes.Buffer
26     if _, err = buf.ReadFrom(b); err != nil {
27         return nil, nil, err
28     }
29     if err = b.Close(); err != nil {
30         return nil, nil, err
31     }
32     return ioutil.NopCloser(&buf), ioutil.NopCloser(byte
33 }
34
35 // dumpConn is a net.Conn which writes to Writer and reads f
36 type dumpConn struct {
37     io.Writer
38     io.Reader
39 }
40
41 func (c *dumpConn) Close() error { ret
```

```

42 func (c *dumpConn) LocalAddr() net.Addr           { ret
43 func (c *dumpConn) RemoteAddr() net.Addr         { ret
44 func (c *dumpConn) SetDeadline(t time.Time) error { ret
45 func (c *dumpConn) SetReadDeadline(t time.Time) error { ret
46 func (c *dumpConn) SetWriteDeadline(t time.Time) error { ret
47
48 // DumpRequestOut is like DumpRequest but includes
49 // headers that the standard http.Transport adds,
50 // such as User-Agent.
51 func DumpRequestOut(req *http.Request, body bool) ([]byte, error) {
52     save := req.Body
53     if !body || req.Body == nil {
54         req.Body = nil
55     } else {
56         var err error
57         save, req.Body, err = drainBody(req.Body)
58         if err != nil {
59             return nil, err
60         }
61     }
62
63     // Since we're using the actual Transport code to write
64     // switch to http so the Transport doesn't try to do
65     // negotiation with our dumpConn and its bytes.Buffer
66     // The wire format for https and http are the same,
67     reqSend := req
68     if req.URL.Scheme == "https" {
69         reqSend = new(http.Request)
70         *reqSend = *req
71         reqSend.URL = new(url.URL)
72         *reqSend.URL = *req.URL
73         reqSend.URL.Scheme = "http"
74     }
75
76     // Use the actual Transport code to record what we write
77     // on the wire, but not using TCP. Use a Transport
78     // customer dialer that returns a fake net.Conn that
79     // for the full input (and recording it), and then return
80     // with a dummy response.
81     var buf bytes.Buffer // records the output
82     pr, pw := io.Pipe()
83     dr := &delegateReader{c: make(chan io.Reader)}
84     // Wait for the request before replying with a dummy
85     go func() {
86         http.ReadRequest(bufio.NewReader(pr))
87         dr.c <- strings.NewReader("HTTP/1.1 204 No Content")
88     }()
89
90     t := &http.Transport{
91         Dial: func(net, addr string) (net.Conn, error) {

```

```

92         return &dumpConn{io.MultiWriter(pw,
93             },
94     }
95
96     _, err := t.RoundTrip(reqSend)
97
98     req.Body = save
99     if err != nil {
100         return nil, err
101     }
102     return buf.Bytes(), nil
103 }
104
105 // delegateReader is a reader that delegates to another read
106 // once it arrives on a channel.
107 type delegateReader struct {
108     c chan io.Reader
109     r io.Reader // nil until received from c
110 }
111
112 func (r *delegateReader) Read(p []byte) (int, error) {
113     if r.r == nil {
114         r.r = <-r.c
115     }
116     return r.r.Read(p)
117 }
118
119 // Return value if nonempty, def otherwise.
120 func valueOrDefault(value, def string) string {
121     if value != "" {
122         return value
123     }
124     return def
125 }
126
127 var reqWriteExcludeHeaderDump = map[string]bool{
128     "Host": true, // not in Header map anyw
129     "Content-Length": true,
130     "Transfer-Encoding": true,
131     "Trailer": true,
132 }
133
134 // dumpAsReceived writes req to w in the form as it was rece
135 // at least as accurately as possible from the information r
136 // the request.
137 func dumpAsReceived(req *http.Request, w io.Writer) error {
138     return nil
139 }
140

```

```

141 // DumpRequest returns the as-received wire representation o
142 // optionally including the request body, for debugging.
143 // DumpRequest is semantically a no-op, but in order to
144 // dump the body, it reads the body data into memory and
145 // changes req.Body to refer to the in-memory copy.
146 // The documentation for http.Request.Write details which fi
147 // of req are used.
148 func DumpRequest(req *http.Request, body bool) (dump []byte,
149     save := req.Body
150     if !body || req.Body == nil {
151         req.Body = nil
152     } else {
153         save, req.Body, err = drainBody(req.Body)
154         if err != nil {
155             return
156         }
157     }
158
159     var b bytes.Buffer
160
161     fmt.Fprintf(&b, "%s %s HTTP/%d.%d\r\n", valueOrDefau
162         req.URL.RequestURI(), req.ProtoMajor, req.Pr
163
164     host := req.Host
165     if host == "" && req.URL != nil {
166         host = req.URL.Host
167     }
168     if host != "" {
169         fmt.Fprintf(&b, "Host: %s\r\n", host)
170     }
171
172     chunked := len(req.TransferEncoding) > 0 && req.Tran
173     if len(req.TransferEncoding) > 0 {
174         fmt.Fprintf(&b, "Transfer-Encoding: %s\r\n",
175     }
176     if req.Close {
177         fmt.Fprintf(&b, "Connection: close\r\n")
178     }
179
180     err = req.Header.WriteSubset(&b, reqWriteExcludeHead
181     if err != nil {
182         return
183     }
184
185     io.WriteString(&b, "\r\n")
186
187     if req.Body != nil {
188         var dest io.Writer = &b
189         if chunked {

```

```

190             dest = NewChunkedWriter(dest)
191         }
192         _, err = io.Copy(dest, req.Body)
193         if chunked {
194             dest.(io.Closer).Close()
195             io.WriteString(&b, "\r\n")
196         }
197     }
198
199     req.Body = save
200     if err != nil {
201         return
202     }
203     dump = b.Bytes()
204     return
205 }
206
207 // DumpResponse is like DumpRequest but dumps a response.
208 func DumpResponse(resp *http.Response, body bool) (dump []byte)
209     var b bytes.Buffer
210     save := resp.Body
211     savecl := resp.ContentLength
212     if !body || resp.Body == nil {
213         resp.Body = nil
214         resp.ContentLength = 0
215     } else {
216         save, resp.Body, err = drainBody(resp.Body)
217         if err != nil {
218             return
219         }
220     }
221     err = resp.Write(&b)
222     resp.Body = save
223     resp.ContentLength = savecl
224     if err != nil {
225         return
226     }
227     dump = b.Bytes()
228     return
229 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/httputil/persist.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package httputil provides HTTP utility functions, complemen
6 // more common ones in the net/http package.
7 package httputil
8
9 import (
10     "bufio"
11     "errors"
12     "io"
13     "net"
14     "net/http"
15     "net/textproto"
16     "sync"
17 )
18
19 var (
20     ErrPersistEOF = &http.ProtocolError{ErrorString: "pe
21     ErrClosed    = &http.ProtocolError{ErrorString: "co
22     ErrPipeline  = &http.ProtocolError{ErrorString: "pi
23 )
24
25 // This is an API usage error - the local side is closed.
26 // ErrPersistEOF (above) reports that the remote side is clo
27 var errClosed = errors.New("i/o operation on closed connecti
28
29 // A ServerConn reads requests and sends responses over an u
30 // connection, until the HTTP keepalive logic commands an en
31 // also allows hijacking the underlying connection by callin
32 // to regain control over the connection. ServerConn support
33 // i.e. requests can be read out of sync (but in the same or
34 // respective responses are sent.
35 //
36 // ServerConn is low-level and should not be needed by most
37 // See Server.
38 type ServerConn struct {
39     lk          sync.Mutex // read-write protects th
40     c           net.Conn
41     r           *bufio.Reader
```

```

42         re, we          error // read/write errors
43         lastbody       io.ReadCloser
44         nread, nwritten int
45         pipereq        map[*http.Request]uint
46
47         pipe textproto.Pipeline
48     }
49
50     // NewServerConn returns a new ServerConn reading and writing
51     // nil, it is the buffer to use when reading c.
52     func NewServerConn(c net.Conn, r *bufio.Reader) *ServerConn
53     {
54         if r == nil {
55             r = bufio.NewReader(c)
56         }
57         return &ServerConn{c: c, r: r, pipereq: make(map[*ht
58     }
59
60     // Hijack detaches the ServerConn and returns the underlying
61     // as the read-side bufio which may have some left over data
62     // called before Read has signaled the end of the keep-alive
63     // should not call Hijack while Read or Write is in progress
64     func (sc *ServerConn) Hijack() (c net.Conn, r *bufio.Reader)
65     {
66         sc.lk.Lock()
67         defer sc.lk.Unlock()
68         c = sc.c
69         r = sc.r
70         sc.c = nil
71         sc.r = nil
72         return
73     }
74
75     // Close calls Hijack and then also closes the underlying co
76     func (sc *ServerConn) Close() error {
77         c, _ := sc.Hijack()
78         if c != nil {
79             return c.Close()
80         }
81         return nil
82     }
83
84     // Read returns the next request on the wire. An ErrPersistE
85     // it is gracefully determined that there are no more reques
86     // first request on an HTTP/1.0 connection, or after a Conne
87     // HTTP/1.1 connection).
88     func (sc *ServerConn) Read() (req *http.Request, err error)
89     {
90         // Ensure ordered execution of Reads and Writes
91         id := sc.pipe.Next()
92         sc.pipe.StartRequest(id)
93         defer func() {

```

```

92         sc.pipe.EndRequest(id)
93         if req == nil {
94             sc.pipe.StartResponse(id)
95             sc.pipe.EndResponse(id)
96         } else {
97             // Remember the pipeline id of this
98             sc.lk.Lock()
99             sc.pipereq[req] = id
100            sc.lk.Unlock()
101        }
102    }()
103
104    sc.lk.Lock()
105    if sc.we != nil { // no point receiving if write-sid
106        defer sc.lk.Unlock()
107        return nil, sc.we
108    }
109    if sc.re != nil {
110        defer sc.lk.Unlock()
111        return nil, sc.re
112    }
113    if sc.r == nil { // connection closed by user in the
114        defer sc.lk.Unlock()
115        return nil, errClosed
116    }
117    r := sc.r
118    lastbody := sc.lastbody
119    sc.lastbody = nil
120    sc.lk.Unlock()
121
122    // Make sure body is fully consumed, even if user do
123    if lastbody != nil {
124        // body.Close is assumed to be idempotent an
125        // it should return the error that its first
126        // returned.
127        err = lastbody.Close()
128        if err != nil {
129            sc.lk.Lock()
130            defer sc.lk.Unlock()
131            sc.re = err
132            return nil, err
133        }
134    }
135
136    req, err = http.ReadRequest(r)
137    sc.lk.Lock()
138    defer sc.lk.Unlock()
139    if err != nil {
140        if err == io.ErrUnexpectedEOF {

```

```

141             // A close from the opposing client
142             // graceful close, even if there was
143             // data before the close.
144             sc.re = ErrPersistEOF
145             return nil, sc.re
146         } else {
147             sc.re = err
148             return req, err
149         }
150     }
151     sc.lastbody = req.Body
152     sc.nread++
153     if req.Close {
154         sc.re = ErrPersistEOF
155         return req, sc.re
156     }
157     return req, err
158 }
159
160 // Pending returns the number of unanswered requests
161 // that have been received on the connection.
162 func (sc *ServerConn) Pending() int {
163     sc.lk.Lock()
164     defer sc.lk.Unlock()
165     return sc.nread - sc.nwritten
166 }
167
168 // Write writes resp in response to req. To close the connec
169 // Response.Close field to true. Write should be considered
170 // it returns an error, regardless of any errors returned on
171 func (sc *ServerConn) Write(req *http.Request, resp *http.Re
172
173     // Retrieve the pipeline ID of this request/response
174     sc.lk.Lock()
175     id, ok := sc.pipereq[req]
176     delete(sc.pipereq, req)
177     if !ok {
178         sc.lk.Unlock()
179         return ErrPipeline
180     }
181     sc.lk.Unlock()
182
183     // Ensure pipeline order
184     sc.pipe.StartResponse(id)
185     defer sc.pipe.EndResponse(id)
186
187     sc.lk.Lock()
188     if sc.we != nil {
189         defer sc.lk.Unlock()

```

```

190         return sc.we
191     }
192     if sc.c == nil { // connection closed by user in the
193         defer sc.lk.Unlock()
194         return ErrClosed
195     }
196     c := sc.c
197     if sc.nread <= sc.nwritten {
198         defer sc.lk.Unlock()
199         return errors.New("persist server pipe count
200     }
201     if resp.Close {
202         // After signaling a keep-alive close, any p
203         // requests will be lost. It is up to the us
204         // before signaling.
205         sc.re = ErrPersistEOF
206     }
207     sc.lk.Unlock()
208
209     err := resp.Write(c)
210     sc.lk.Lock()
211     defer sc.lk.Unlock()
212     if err != nil {
213         sc.we = err
214         return err
215     }
216     sc.nwritten++
217
218     return nil
219 }
220
221 // A ClientConn sends request and receives headers over an u
222 // connection, while respecting the HTTP keepalive logic. Cl
223 // supports hijacking the connection calling Hijack to
224 // regain control of the underlying net.Conn and deal with i
225 //
226 // ClientConn is low-level and should not be needed by most
227 // See Client.
228 type ClientConn struct {
229     lk          sync.Mutex // read-write protects th
230     c           net.Conn
231     r           *bufio.Reader
232     re, we      error // read/write errors
233     lastbody    io.ReadCloser
234     nread, nwritten int
235     pipereq     map[*http.Request]uint
236
237     pipe      textproto.Pipeline
238     writeReq func(*http.Request, io.Writer) error
239 }

```

```

240
241 // NewClientConn returns a new ClientConn reading and writin
242 // nil, it is the buffer to use when reading c.
243 func NewClientConn(c net.Conn, r *bufio.Reader) *ClientConn
244     if r == nil {
245         r = bufio.NewReader(c)
246     }
247     return &ClientConn{
248         c:      c,
249         r:      r,
250         pipereq: make(map[*http.Request]uint),
251         writeReq: (*http.Request).Write,
252     }
253 }
254
255 // NewProxyClientConn works like NewClientConn but writes Re
256 // using Request's WriteProxy method.
257 func NewProxyClientConn(c net.Conn, r *bufio.Reader) *Client
258     cc := NewClientConn(c, r)
259     cc.writeReq = (*http.Request).WriteProxy
260     return cc
261 }
262
263 // Hijack detaches the ClientConn and returns the underlying
264 // as the read-side bufio which may have some left over data
265 // called before the user or Read have signaled the end of t
266 // logic. The user should not call Hijack while Read or Writ
267 func (cc *ClientConn) Hijack() (c net.Conn, r *bufio.Reader)
268     cc.lk.Lock()
269     defer cc.lk.Unlock()
270     c = cc.c
271     r = cc.r
272     cc.c = nil
273     cc.r = nil
274     return
275 }
276
277 // Close calls Hijack and then also closes the underlying co
278 func (cc *ClientConn) Close() error {
279     c, _ := cc.Hijack()
280     if c != nil {
281         return c.Close()
282     }
283     return nil
284 }
285
286 // Write writes a request. An ErrPersistEOF error is returne
287 // has been closed in an HTTP keepalive sense. If req.Close
288 // keepalive connection is logically closed after this requ

```

```

289 // server is informed. An ErrUnexpectedEOF indicates the rem
290 // underlying TCP connection, which is usually considered as
291 func (cc *ClientConn) Write(req *http.Request) (err error) {
292
293     // Ensure ordered execution of Writes
294     id := cc.pipe.Next()
295     cc.pipe.StartRequest(id)
296     defer func() {
297         cc.pipe.EndRequest(id)
298         if err != nil {
299             cc.pipe.StartResponse(id)
300             cc.pipe.EndResponse(id)
301         } else {
302             // Remember the pipeline id of this
303             cc.lk.Lock()
304             cc.pipereq[req] = id
305             cc.lk.Unlock()
306         }
307     }()
308
309     cc.lk.Lock()
310     if cc.re != nil { // no point sending if read-side c
311         defer cc.lk.Unlock()
312         return cc.re
313     }
314     if cc.we != nil {
315         defer cc.lk.Unlock()
316         return cc.we
317     }
318     if cc.c == nil { // connection closed by user in the
319         defer cc.lk.Unlock()
320         return errClosed
321     }
322     c := cc.c
323     if req.Close {
324         // We write the EOF to the write-side error,
325         // still might be some pipelined reads
326         cc.we = ErrPersistEOF
327     }
328     cc.lk.Unlock()
329
330     err = cc.writeReq(req, c)
331     cc.lk.Lock()
332     defer cc.lk.Unlock()
333     if err != nil {
334         cc.we = err
335         return err
336     }
337     cc.nwritten++

```

```

338
339     return nil
340 }
341
342 // Pending returns the number of unanswered requests
343 // that have been sent on the connection.
344 func (cc *ClientConn) Pending() int {
345     cc.lk.Lock()
346     defer cc.lk.Unlock()
347     return cc.nwritten - cc.nread
348 }
349
350 // Read reads the next response from the wire. A valid respo
351 // returned together with an ErrPersistEOF, which means that
352 // requested that this be the last request serviced. Read ca
353 // concurrently with Write, but not with another Read.
354 func (cc *ClientConn) Read(req *http.Request) (resp *http.Re
355     // Retrieve the pipeline ID of this request/response
356     cc.lk.Lock()
357     id, ok := cc.pipereq[req]
358     delete(cc.pipereq, req)
359     if !ok {
360         cc.lk.Unlock()
361         return nil, ErrPipeline
362     }
363     cc.lk.Unlock()
364
365     // Ensure pipeline order
366     cc.pipe.StartResponse(id)
367     defer cc.pipe.EndResponse(id)
368
369     cc.lk.Lock()
370     if cc.re != nil {
371         defer cc.lk.Unlock()
372         return nil, cc.re
373     }
374     if cc.r == nil { // connection closed by user in the
375         defer cc.lk.Unlock()
376         return nil, errClosed
377     }
378     r := cc.r
379     lastbody := cc.lastbody
380     cc.lastbody = nil
381     cc.lk.Unlock()
382
383     // Make sure body is fully consumed, even if user do
384     if lastbody != nil {
385         // body.Close is assumed to be idempotent an
386         // it should return the error that its first
387         // returned.

```

```

388         err = lastbody.Close()
389         if err != nil {
390             cc.lk.Lock()
391             defer cc.lk.Unlock()
392             cc.re = err
393             return nil, err
394         }
395     }
396
397     resp, err = http.ReadResponse(r, req)
398     cc.lk.Lock()
399     defer cc.lk.Unlock()
400     if err != nil {
401         cc.re = err
402         return resp, err
403     }
404     cc.lastbody = resp.Body
405
406     cc.nread++
407
408     if resp.Close {
409         cc.re = ErrPersistEOF // don't send any more
410         return resp, cc.re
411     }
412     return resp, err
413 }
414
415 // Do is convenience method that writes a request and reads
416 func (cc *ClientConn) Do(req *http.Request) (resp *http.Resp
417     err = cc.Write(req)
418     if err != nil {
419         return
420     }
421     return cc.Read(req)
422 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/httputil/reverseproxy.

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // HTTP reverse proxy handler
6
7 package httputil
8
9 import (
10     "io"
11     "log"
12     "net"
13     "net/http"
14     "net/url"
15     "strings"
16     "sync"
17     "time"
18 )
19
20 // ReverseProxy is an HTTP Handler that takes an incoming re
21 // sends it to another server, proxying the response back to
22 // client.
23 type ReverseProxy struct {
24     // Director must be a function which modifies
25     // the request into a new request to be sent
26     // using Transport. Its response is then copied
27     // back to the original client unmodified.
28     Director func(*http.Request)
29
30     // The transport used to perform proxy requests.
31     // If nil, http.DefaultTransport is used.
32     Transport http.RoundTripper
33
34     // FlushInterval specifies the flush interval
35     // to flush to the client while copying the
36     // response body.
37     // If zero, no periodic flushing is done.
38     FlushInterval time.Duration
39 }
40
41 func singleJoiningSlash(a, b string) string {
```

```

42     aslash := strings.HasSuffix(a, "/")
43     bslash := strings.HasPrefix(b, "/")
44     switch {
45     case aslash && bslash:
46         return a + b[1:]
47     case !aslash && !bslash:
48         return a + "/" + b
49     }
50     return a + b
51 }
52
53 // NewSingleHostReverseProxy returns a new ReverseProxy that
54 // URLs to the scheme, host, and base path provided in target
55 // target's path is "/base" and the incoming request was for
56 // the target request will be for /base/dir.
57 func NewSingleHostReverseProxy(target *url.URL) *ReverseProxy {
58     targetQuery := target.RawQuery
59     director := func(req *http.Request) {
60         req.URL.Scheme = target.Scheme
61         req.URL.Host = target.Host
62         req.URL.Path = singleJoiningSlash(target.Path, req.URL.Path)
63         if targetQuery == "" || req.URL.RawQuery == "" {
64             req.URL.RawQuery = targetQuery + req.URL.RawQuery
65         } else {
66             req.URL.RawQuery = targetQuery + "&" + req.URL.RawQuery
67         }
68     }
69     return &ReverseProxy{Director: director}
70 }
71
72 func copyHeader(dst, src http.Header) {
73     for k, vv := range src {
74         for _, v := range vv {
75             dst.Add(k, v)
76         }
77     }
78 }
79
80 func (p *ReverseProxy) ServeHTTP(rw http.ResponseWriter, req *http.Request) {
81     transport := p.Transport
82     if transport == nil {
83         transport = http.DefaultTransport
84     }
85
86     outreq := new(http.Request)
87     *outreq = *req // includes shallow copies of maps, b
88
89     p.Director(outreq)
90     outreq.Proto = "HTTP/1.1"
91     outreq.ProtoMajor = 1

```

```

92         outreq.ProtoMinor = 1
93         outreq.Close = false
94
95         // Remove the connection header to the backend. We
96         // persistent connection, regardless of what the cli
97         // to us. This is modifying the same underlying map
98         // (shallow copied above) so we only copy it if nece
99         if outreq.Header.Get("Connection") != "" {
100             outreq.Header = make(http.Header)
101             copyHeader(outreq.Header, req.Header)
102             outreq.Header.Del("Connection")
103         }
104
105         if clientIp, _, err := net.SplitHostPort(req.RemoteA
106             outreq.Header.Set("X-Forwarded-For", clientI
107         }
108
109         res, err := transport.RoundTrip(outreq)
110         if err != nil {
111             log.Printf("http: proxy error: %v", err)
112             rw.WriteHeader(http.StatusInternalServerErrorErro
113             return
114         }
115
116         copyHeader(rw.Header(), res.Header)
117
118         rw.WriteHeader(res.StatusCode)
119
120         if res.Body != nil {
121             var dst io.Writer = rw
122             if p.FlushInterval != 0 {
123                 if wf, ok := rw.(writeFlusher); ok {
124                     dst = &maxLatencyWriter{dst:
125                         }
126                 }
127             }
128             io.Copy(dst, res.Body)
129         }
130     }
131     type writeFlusher interface {
132         io.Writer
133         http.Flusher
134     }
135
136     type maxLatencyWriter struct {
137         dst      writeFlusher
138         latency time.Duration
139
140         lk      sync.Mutex // protects init of done, as well wr

```

```

141         done chan bool
142     }
143
144     func (m *maxLatencyWriter) Write(p []byte) (n int, err error)
145         m.lk.Lock()
146         defer m.lk.Unlock()
147         if m.done == nil {
148             m.done = make(chan bool)
149             go m.flushLoop()
150         }
151         n, err = m.dst.Write(p)
152         if err != nil {
153             m.done <- true
154         }
155         return
156     }
157
158     func (m *maxLatencyWriter) flushLoop() {
159         t := time.NewTicker(m.latency)
160         defer t.Stop()
161         for {
162             select {
163             case <-t.C:
164                 m.lk.Lock()
165                 m.dst.Flush()
166                 m.lk.Unlock()
167             case <-m.done:
168                 return
169             }
170         }
171         panic("unreached")
172     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/http/pprof/pprof.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package pprof serves via its HTTP server runtime profilin
6 // in the format expected by the pprof visualization tool.
7 // For more information about pprof, see
8 // http://code.google.com/p/google-perftools/.
9 //
10 // The package is typically only imported for the side effect
11 // registering its HTTP handlers.
12 // The handled paths all begin with /debug/pprof/.
13 //
14 // To use pprof, link this package into your program:
15 //     import _ "net/http/pprof"
16 //
17 // Then use the pprof tool to look at the heap profile:
18 //
19 //     go tool pprof http://localhost:6060/debug/pprof/heap
20 //
21 // Or to look at a 30-second CPU profile:
22 //
23 //     go tool pprof http://localhost:6060/debug/pprof/profile
24 //
25 // Or to view all available profiles:
26 //
27 //     go tool pprof http://localhost:6060/debug/pprof/
28 //
29 // For a study of the facility in action, visit
30 //
31 //     http://blog.golang.org/2011/06/profiling-go-programs
32 //
33 package pprof
34
35 import (
36     "bufio"
37     "bytes"
38     "fmt"
39     "html/template"
40     "io"
41     "log"
```

```

42     "net/http"
43     "os"
44     "runtime"
45     "runtime/pprof"
46     "strconv"
47     "strings"
48     "time"
49 )
50
51 func init() {
52     http.Handle("/debug/pprof/", http.HandlerFunc(Index))
53     http.Handle("/debug/pprof/cmdline", http.HandlerFunc(Cmdline))
54     http.Handle("/debug/pprof/profile", http.HandlerFunc(Profile))
55     http.Handle("/debug/pprof/symbol", http.HandlerFunc(Symbol))
56 }
57
58 // Cmdline responds with the running program's
59 // command line, with arguments separated by NUL bytes.
60 // The package initialization registers it as /debug/pprof/c
61 func Cmdline(w http.ResponseWriter, r *http.Request) {
62     w.Header().Set("Content-Type", "text/plain; charset=
63     fmt.Fprintf(w, strings.Join(os.Args, "\x00"))
64 }
65
66 // Profile responds with the pprof-formatted cpu profile.
67 // The package initialization registers it as /debug/pprof/p
68 func Profile(w http.ResponseWriter, r *http.Request) {
69     sec, _ := strconv.ParseInt(r.FormValue("seconds"), 1
70     if sec == 0 {
71         sec = 30
72     }
73
74     // Set Content Type assuming StartCPUProfile will wo
75     // because if it does it starts writing.
76     w.Header().Set("Content-Type", "application/octet-st
77     if err := pprof.StartCPUProfile(w); err != nil {
78         // StartCPUProfile failed, so no writes yet.
79         // Can change header back to text content
80         // and send error code.
81         w.Header().Set("Content-Type", "text/plain;
82         w.WriteHeader(http.StatusInternalServerError)
83         fmt.Fprintf(w, "Could not enable CPU profili
84         return
85     }
86     time.Sleep(time.Duration(sec) * time.Second)
87     pprof.StopCPUProfile()
88 }
89
90 // Symbol looks up the program counters listed in the reques
91 // responding with a table mapping program counters to funct

```

```

92 // The package initialization registers it as /debug/pprof/s
93 func Symbol(w http.ResponseWriter, r *http.Request) {
94     w.Header().Set("Content-Type", "text/plain; charset="
95
96     // We have to read the whole POST body before
97     // writing any output. Buffer the output here.
98     var buf bytes.Buffer
99
100    // We don't know how many symbols we have, but we
101    // do have symbol information. Pprof only cares whe
102    // this number is 0 (no symbols available) or > 0.
103    fmt.Fprintf(&buf, "num_symbols: 1\n")
104
105    var b *bufio.Reader
106    if r.Method == "POST" {
107        b = bufio.NewReader(r.Body)
108    } else {
109        b = bufio.NewReader(strings.NewReader(r.URL.
110    })
111
112    for {
113        word, err := b.ReadSlice('+')
114        if err == nil {
115            word = word[0 : len(word)-1] // trim
116        }
117        pc, _ := strconv.ParseUint(string(word), 0,
118        if pc != 0 {
119            f := runtime.FuncForPC(uintptr(pc))
120            if f != nil {
121                fmt.Fprintf(&buf, "%#x %s\n"
122            }
123        }
124
125        // Wait until here to check for err; the las
126        // symbol will have an err because it doesn'
127        if err != nil {
128            if err != io.EOF {
129                fmt.Fprintf(&buf, "reading r
130            }
131            break
132        }
133    }
134
135    w.Write(buf.Bytes())
136 }
137
138 // Handler returns an HTTP handler that serves the named pro
139 func Handler(name string) http.Handler {
140     return handler(name)

```

```

141 }
142
143 type handler string
144
145 func (name handler) ServeHTTP(w http.ResponseWriter, r *http
146     w.Header().Set("Content-Type", "text/plain; charset=
147     debug, _ := strconv.Atoi(r.FormValue("debug"))
148     p := pprof.Lookup(string(name))
149     if p == nil {
150         w.WriteHeader(404)
151         fmt.Fprintf(w, "Unknown profile: %s\n", name)
152         return
153     }
154     p.WriteTo(w, debug)
155     return
156 }
157
158 // Index responds with the pprof-formatted profile named by
159 // For example, "/debug/pprof/heap" serves the "heap" profil
160 // Index responds to a request for "/debug/pprof/" with an H
161 // listing the available profiles.
162 func Index(w http.ResponseWriter, r *http.Request) {
163     if strings.HasPrefix(r.URL.Path, "/debug/pprof/") {
164         name := r.URL.Path[len("/debug/pprof/"):]
165         if name != "" {
166             handler(name).ServeHTTP(w, r)
167             return
168         }
169     }
170
171     profiles := pprof.Profiles()
172     if err := indexTmpl.Execute(w, profiles); err != nil
173         log.Print(err)
174     }
175 }
176
177 var indexTmpl = template.Must(template.New("index").Parse(`<
178 <head>
179 <title>/debug/pprof/</title>
180 </head>
181 /debug/pprof/<br>
182 <br>
183 <body>
184 profiles:<br>
185 <table>
186 {{range .}}
187 <tr><td align=right>{{.Count}}<td><a href="/debug/pprof/{{.N
188 {{end}}
189 </table>

```

```
190 <br>
191 <a href="/debug/pprof/goroutine?debug=2">full goroutine stac
192 </body>
193 </html>
194 `))
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/mail/message.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6 Package mail implements parsing of mail messages.
7
8 For the most part, this package follows the syntax as specif
9 Notable divergences:
10     * Obsolete address formats are not parsed, including
11     embedded route information.
12     * Group addresses are not parsed.
13     * The full range of spacing (the CFWS syntax element
14     such as breaking addresses across lines.
15 */
16 package mail
17
18 import (
19     "bufio"
20     "bytes"
21     "encoding/base64"
22     "errors"
23     "fmt"
24     "io"
25     "io/ioutil"
26     "log"
27     "net/textproto"
28     "strconv"
29     "strings"
30     "time"
31 )
32
33 var debug = debugT(false)
34
35 type debugT bool
36
37 func (d debugT) Printf(format string, args ...interface{}) {
38     if d {
39         log.Printf(format, args...)
40     }
41 }
```

```

42
43 // A Message represents a parsed mail message.
44 type Message struct {
45     Header Header
46     Body   io.Reader
47 }
48
49 // ReadMessage reads a message from r.
50 // The headers are parsed, and the body of the message will
51 func ReadMessage(r io.Reader) (msg *Message, err error) {
52     tp := textproto.NewReader(bufio.NewReader(r))
53
54     hdr, err := tp.ReadMIMEHeader()
55     if err != nil {
56         return nil, err
57     }
58
59     return &Message{
60         Header: Header(hdr),
61         Body:   tp.R,
62     }, nil
63 }
64
65 // Layouts suitable for passing to time.Parse.
66 // These are tried in order.
67 var dateLayouts []string
68
69 func init() {
70     // Generate layouts based on RFC 5322, section 3.3.
71
72     dows := [...]string{"", "Mon, "} // day-of-week
73     days := [...]string{"2", "02"} // day = 1*2DIG
74     years := [...]string{"2006", "06"} // year = 4*DIG
75     seconds := [...]string{":05", ""} // second
76     zones := [...]string{"-0700", "MST"} // zone = (("+"
77
78     for _, dow := range dows {
79         for _, day := range days {
80             for _, year := range years {
81                 for _, second := range secon
82                     for _, zone := range
83                         s := dow + d
84                             dateLayouts
85                             }
86                             }
87                             }
88                             }
89                             }
90 }
91

```

```

92 func parseDate(date string) (time.Time, error) {
93     for _, layout := range dateLayouts {
94         t, err := time.Parse(layout, date)
95         if err == nil {
96             return t, nil
97         }
98     }
99     return time.Time{}, errors.New("mail: header could n
100 }
101
102 // A Header represents the key-value pairs in a mail message
103 type Header map[string][]string
104
105 // Get gets the first value associated with the given key.
106 // If there are no values associated with the key, Get retur
107 func (h Header) Get(key string) string {
108     return textproto.MIMEHeader(h).Get(key)
109 }
110
111 var ErrHeaderNotPresent = errors.New("mail: header not in me
112
113 // Date parses the Date header field.
114 func (h Header) Date() (time.Time, error) {
115     hdr := h.Get("Date")
116     if hdr == "" {
117         return time.Time{}, ErrHeaderNotPresent
118     }
119     return parseDate(hdr)
120 }
121
122 // AddressList parses the named header field as a list of ad
123 func (h Header) AddressList(key string) ([]*Address, error)
124     hdr := h.Get(key)
125     if hdr == "" {
126         return nil, ErrHeaderNotPresent
127     }
128     return newAddrParser(hdr).parseAddressList()
129 }
130
131 // Address represents a single mail address.
132 // An address such as "Barry Gibbs <bg@example.com>" is repr
133 // as Address{Name: "Barry Gibbs", Address: "bg@example.com"
134 type Address struct {
135     Name    string // Proper name; may be empty.
136     Address string // user@domain
137 }
138
139 // String formats the address as a valid RFC 5322 address.
140 // If the address's name contains non-ASCII characters

```

```

141 // the name will be rendered according to RFC 2047.
142 func (a *Address) String() string {
143     s := "<" + a.Address + ">"
144     if a.Name == "" {
145         return s
146     }
147     // If every character is printable ASCII, quoting is
148     allPrintable := true
149     for i := 0; i < len(a.Name); i++ {
150         if !isVchar(a.Name[i]) {
151             allPrintable = false
152             break
153         }
154     }
155     if allPrintable {
156         b := bytes.NewBufferString(`"`)
157         for i := 0; i < len(a.Name); i++ {
158             if !isQtext(a.Name[i]) {
159                 b.WriteByte(`\`)
160             }
161             b.WriteByte(a.Name[i])
162         }
163         b.WriteString(`" `)
164         b.WriteString(s)
165         return b.String()
166     }
167
168     // UTF-8 "Q" encoding
169     b := bytes.NewBufferString("=?utf-8?q?")
170     for i := 0; i < len(a.Name); i++ {
171         switch c := a.Name[i]; {
172             case c == ' ':
173                 b.WriteByte('_')
174             case isVchar(c) && c != '=' && c != '?' && c
175                 b.WriteByte(c)
176             default:
177                 fmt.Fprintf(b, "%02X", c)
178         }
179     }
180     b.WriteString("=? ")
181     b.WriteString(s)
182     return b.String()
183 }
184
185 type addrParser []byte
186
187 func newAddrParser(s string) *addrParser {
188     p := addrParser(s)
189     return &p

```

```

190 }
191
192 func (p *addrParser) parseAddressList() ([]*Address, error)
193     var list []*Address
194     for {
195         p.skipSpace()
196         addr, err := p.parseAddress()
197         if err != nil {
198             return nil, err
199         }
200         list = append(list, addr)
201
202         p.skipSpace()
203         if p.empty() {
204             break
205         }
206         if !p.consume(',') {
207             return nil, errors.New("mail: expect
208         }
209     }
210     return list, nil
211 }
212
213 // parseAddress parses a single RFC 5322 address at the star
214 func (p *addrParser) parseAddress() (addr *Address, err error)
215     debug.Printf("parseAddress: %q", *p)
216     p.skipSpace()
217     if p.empty() {
218         return nil, errors.New("mail: no address")
219     }
220
221     // address = name-addr / addr-spec
222     // TODO(dsymonds): Support parsing group address.
223
224     // addr-spec has a more restricted grammar than name
225     // so try parsing it first, and fallback to name-addr
226     // TODO(dsymonds): Is this really correct?
227     spec, err := p.consumeAddrSpec()
228     if err == nil {
229         return &Address{
230             Address: spec,
231         }, err
232     }
233     debug.Printf("parseAddress: not an addr-spec: %v", e
234     debug.Printf("parseAddress: state is now %q", *p)
235
236     // display-name
237     var displayName string
238     if p.peek() != '<' {
239         displayName, err = p.consumePhrase()

```

```

240         if err != nil {
241             return nil, err
242         }
243     }
244     debug.Printf("parseAddress: displayName=%q", display
245
246     // angle-addr = "<" addr-spec ">"
247     p.skipSpace()
248     if !p.consume('<') {
249         return nil, errors.New("mail: no angle-addr"
250     }
251     spec, err = p.consumeAddrSpec()
252     if err != nil {
253         return nil, err
254     }
255     if !p.consume('>') {
256         return nil, errors.New("mail: unclosed angle
257     }
258     debug.Printf("parseAddress: spec=%q", spec)
259
260     return &Address{
261         Name:    displayName,
262         Address: spec,
263     }, nil
264 }
265
266 // consumeAddrSpec parses a single RFC 5322 addr-spec at the
267 func (p *addrParser) consumeAddrSpec() (spec string, err error) {
268     debug.Printf("consumeAddrSpec: %q", *p)
269
270     orig := *p
271     defer func() {
272         if err != nil {
273             *p = orig
274         }
275     }()
276
277     // local-part = dot-atom / quoted-string
278     var localPart string
279     p.skipSpace()
280     if p.empty() {
281         return "", errors.New("mail: no addr-spec")
282     }
283     if p.peek() == '"' {
284         // quoted-string
285         debug.Printf("consumeAddrSpec: parsing quote
286             localPart, err = p.consumeQuotedString()
287     } else {
288         // dot-atom

```

```

289         debug.Printf("consumeAddrSpec: parsing dot-a
290         localPart, err = p.consumeAtom(true)
291     }
292     if err != nil {
293         debug.Printf("consumeAddrSpec: failed: %v",
294         return "", err
295     }
296
297     if !p.consume('@') {
298         return "", errors.New("mail: missing @ in ad
299     }
300
301     // domain = dot-atom / domain-literal
302     var domain string
303     p.skipSpace()
304     if p.empty() {
305         return "", errors.New("mail: no domain in ad
306     }
307     // TODO(dsymonds): Handle domain-literal
308     domain, err = p.consumeAtom(true)
309     if err != nil {
310         return "", err
311     }
312
313     return localPart + "@" + domain, nil
314 }
315
316 // consumePhrase parses the RFC 5322 phrase at the start of
317 func (p *addrParser) consumePhrase() (phrase string, err error) {
318     debug.Printf("consumePhrase: [%s]", *p)
319     // phrase = 1*word
320     var words []string
321     for {
322         // word = atom / quoted-string
323         var word string
324         p.skipSpace()
325         if p.empty() {
326             return "", errors.New("mail: missing
327         }
328         if p.peek() == '"' {
329             // quoted-string
330             word, err = p.consumeQuotedString()
331         } else {
332             // atom
333             word, err = p.consumeAtom(false)
334         }
335
336         // RFC 2047 encoded-word starts with =?, end
337         if err == nil && strings.HasPrefix(word, "=?"

```

```

338             word, err = decodeRFC2047Word(word)
339         }
340
341         if err != nil {
342             break
343         }
344         debug.Printf("consumePhrase: consumed %q", w
345         words = append(words, word)
346     }
347     // Ignore any error if we got at least one word.
348     if err != nil && len(words) == 0 {
349         debug.Printf("consumePhrase: hit err: %v", e
350         return "", errors.New("mail: missing word in
351     }
352     phrase = strings.Join(words, " ")
353     return phrase, nil
354 }
355
356 // consumeQuotedString parses the quoted string at the start
357 func (p *addrParser) consumeQuotedString() (qs string, err e
358     // Assume first byte is '"'.
359     i := 1
360     qsb := make([]byte, 0, 10)
361 Loop:
362     for {
363         if i >= p.len() {
364             return "", errors.New("mail: unclosed
365         }
366         switch c := (*p)[i]; {
367         case c == '"':
368             break Loop
369         case c == '\\':
370             if i+1 == p.len() {
371                 return "", errors.New("mail:
372             }
373             qsb = append(qsb, (*p)[i+1])
374             i += 2
375         case isQtext(c), c == ' ' || c == '\t':
376             // qtext (printable US-ASCII excludi
377             // FWS (almost; we're ignoring CRLF)
378             qsb = append(qsb, c)
379             i++
380         default:
381             return "", fmt.Errorf("mail: bad cha
382         }
383     }
384     *p = (*p)[i+1:]
385     return string(qsb), nil
386 }
387

```

```

388 // consumeAtom parses an RFC 5322 atom at the start of p.
389 // If dot is true, consumeAtom parses an RFC 5322 dot-atom i
390 func (p *addrParser) consumeAtom(dot bool) (atom string, err
391     if !isAtext(p.peek(), false) {
392         return "", errors.New("mail: invalid string"
393     }
394     i := 1
395     for ; i < p.len() && isAtext((*p)[i], dot); i++ {
396     }
397     atom, *p = string((*p)[:i]), (*p)[i:]
398     return atom, nil
399 }
400
401 func (p *addrParser) consume(c byte) bool {
402     if p.empty() || p.peek() != c {
403         return false
404     }
405     *p = (*p)[1:]
406     return true
407 }
408
409 // skipSpace skips the leading space and tab characters.
410 func (p *addrParser) skipSpace() {
411     *p = bytes.TrimLeft(*p, " \t")
412 }
413
414 func (p *addrParser) peek() byte {
415     return (*p)[0]
416 }
417
418 func (p *addrParser) empty() bool {
419     return p.len() == 0
420 }
421
422 func (p *addrParser) len() int {
423     return len(*p)
424 }
425
426 func decodeRFC2047Word(s string) (string, error) {
427     fields := strings.Split(s, "?")
428     if len(fields) != 5 || fields[0] != "=" || fields[4]
429         return "", errors.New("mail: address not RFC
430     }
431     charset, enc := strings.ToLower(fields[1]), strings.
432     if charset != "iso-8859-1" && charset != "utf-8" {
433         return "", fmt.Errorf("mail: charset not sup
434     }
435
436     in := bytes.NewBufferString(fields[3])

```

```

437     var r io.Reader
438     switch enc {
439     case "b":
440         r = base64.NewDecoder(base64.StdEncoding, in
441     case "q":
442         r = qDecoder{r: in}
443     default:
444         return "", fmt.Errorf("mail: RFC 2047 encodi
445     }
446
447     dec, err := ioutil.ReadAll(r)
448     if err != nil {
449         return "", err
450     }
451
452     switch charset {
453     case "iso-8859-1":
454         b := new(bytes.Buffer)
455         for _, c := range dec {
456             b.WriteRune(rune(c))
457         }
458         return b.String(), nil
459     case "utf-8":
460         return string(dec), nil
461     }
462     panic("unreachable")
463 }
464
465 type qDecoder struct {
466     r      io.Reader
467     scratch [2]byte
468 }
469
470 func (qd qDecoder) Read(p []byte) (n int, err error) {
471     // This method writes at most one byte into p.
472     if len(p) == 0 {
473         return 0, nil
474     }
475     if _, err := qd.r.Read(qd.scratch[:1]); err != nil {
476         return 0, err
477     }
478     switch c := qd.scratch[0]; {
479     case c == '=':
480         if _, err := io.ReadFull(qd.r, qd.scratch[:2]
481         return 0, err
482     }
483     x, err := strconv.ParseInt(string(qd.scratch
484     if err != nil {
485         return 0, fmt.Errorf("mail: invalid

```

```

486         }
487         p[0] = byte(x)
488     case c == '_':
489         p[0] = ' '
490     default:
491         p[0] = c
492     }
493     return 1, nil
494 }
495
496 var atextChars = []byte("ABCDEFGHJKLMNOPQRSTUVWXYZ" +
497     "abcdefghijklmnopqrstuvwxyz" +
498     "0123456789" +
499     "!#$%&'*+,-/=/?^_`{|}~")
500
501 // isAtext returns true if c is an RFC 5322 atext character.
502 // If dot is true, period is included.
503 func isAtext(c byte, dot bool) bool {
504     if dot && c == '.' {
505         return true
506     }
507     return bytes.IndexByte(atextChars, c) >= 0
508 }
509
510 // isQtext returns true if c is an RFC 5322 qtext character.
511 func isQtext(c byte) bool {
512     // Printable US-ASCII, excluding backslash or quote.
513     if c == '\\' || c == '"' {
514         return false
515     }
516     return '!' <= c && c <= '~'
517 }
518
519 // isVchar returns true if c is an RFC 5322 VCHAR character.
520 func isVchar(c byte) bool {
521     // Visible (printing) characters.
522     return '!' <= c && c <= '~'
523 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/rpc/client.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package rpc
6
7 import (
8     "bufio"
9     "encoding/gob"
10    "errors"
11    "io"
12    "log"
13    "net"
14    "net/http"
15    "sync"
16 )
17
18 // ServerError represents an error that has been returned fr
19 // the remote side of the RPC connection.
20 type ServerError string
21
22 func (e ServerError) Error() string {
23     return string(e)
24 }
25
26 var ErrShutdown = errors.New("connection is shut down")
27
28 // Call represents an active RPC.
29 type Call struct {
30     ServiceMethod string // The name of the service
31     Args           interface{} // The argument to the fun
32     Reply          interface{} // The reply from the func
33     Error          error      // After completion, the e
34     Done          chan *Call // Strobes when call is co
35 }
36
37 // Client represents an RPC Client.
38 // There may be multiple outstanding Calls associated
39 // with a single Client, and a Client may be used by
40 // multiple goroutines simultaneously.
41 type Client struct {
42     mutex    sync.Mutex // protects pending, seq, reques
43     sending sync.Mutex
44     request Request
```

```

45         seq      uint64
46         codec    ClientCodec
47         pending  map[uint64]*Call
48         closing  bool
49         shutdown bool
50     }
51
52     // A ClientCodec implements writing of RPC requests and
53     // reading of RPC responses for the client side of an RPC se
54     // The client calls WriteRequest to write a request to the c
55     // and calls ReadResponseHeader and ReadResponseBody in pair
56     // to read responses. The client calls Close when finished
57     // connection. ReadResponseBody may be called with a nil
58     // argument to force the body of the response to be read and
59     // discarded.
60     type ClientCodec interface {
61         WriteRequest(*Request, interface{}) error
62         ReadResponseHeader(*Response) error
63         ReadResponseBody(interface{}) error
64
65         Close() error
66     }
67
68     func (client *Client) send(call *Call) {
69         client.sending.Lock()
70         defer client.sending.Unlock()
71
72         // Register this call.
73         client.mutex.Lock()
74         if client.shutdown {
75             call.Error = ErrShutdown
76             client.mutex.Unlock()
77             call.done()
78             return
79         }
80         seq := client.seq
81         client.seq++
82         client.pending[seq] = call
83         client.mutex.Unlock()
84
85         // Encode and send the request.
86         client.request.Seq = seq
87         client.request.ServiceMethod = call.ServiceMethod
88         err := client.codec.WriteRequest(&client.request, ca
89         if err != nil {
90             client.mutex.Lock()
91             delete(client.pending, seq)
92             client.mutex.Unlock()
93             call.Error = err
94             call.done()

```

```

95     }
96 }
97
98 func (client *Client) input() {
99     var err error
100    var response Response
101    for err == nil {
102        response = Response{}
103        err = client.codec.ReadResponseHeader(&respo
104        if err != nil {
105            if err == io.EOF && !client.closing
106                err = io.ErrUnexpectedEOF
107        }
108        break
109    }
110    seq := response.Seq
111    client.mutex.Lock()
112    call := client.pending[seq]
113    delete(client.pending, seq)
114    client.mutex.Unlock()
115
116    if response.Error == "" {
117        err = client.codec.ReadResponseBody(
118        if err != nil {
119            call.Error = errors.New("rea
120        }
121    } else {
122        // We've got an error response. Give
123        // any subsequent requests will get
124        // error if there is one.
125        call.Error = ServerError(response.Er
126        err = client.codec.ReadResponseBody(
127        if err != nil {
128            err = errors.New("reading er
129        }
130    }
131    call.done()
132 }
133 // Terminate pending calls.
134 client.sending.Lock()
135 client.mutex.Lock()
136 client.shutdown = true
137 closing := client.closing
138 for _, call := range client.pending {
139     call.Error = err
140     call.done()
141 }
142 client.mutex.Unlock()
143 client.sending.Unlock()

```

```

144         if err != io.EOF && !closing {
145             log.Println("rpc: client protocol error:", e)
146         }
147     }
148
149     func (call *Call) done() {
150         select {
151         case call.Done <- call:
152             // ok
153         default:
154             // We don't want to block here. It is the c
155             // sure the channel has enough buffer space.
156             log.Println("rpc: discarding Call reply due
157         }
158     }
159
160     // NewClient returns a new Client to handle requests to the
161     // set of services at the other end of the connection.
162     // It adds a buffer to the write side of the connection so
163     // the header and payload are sent as a unit.
164     func NewClient(conn io.ReadWriteCloser) *Client {
165         encBuf := bufio.NewWriter(conn)
166         client := &gobClientCodec{conn, gob.NewDecoder(conn)
167         return NewClientWithCodec(client)
168     }
169
170     // NewClientWithCodec is like NewClient but uses the specifi
171     // codec to encode requests and decode responses.
172     func NewClientWithCodec(codec ClientCodec) *Client {
173         client := &Client{
174             codec:    codec,
175             pending:  make(map[uint64]*Call),
176         }
177         go client.input()
178         return client
179     }
180
181     type gobClientCodec struct {
182         rwc    io.ReadWriteCloser
183         dec    *gob.Decoder
184         enc    *gob.Encoder
185         encBuf *bufio.Writer
186     }
187
188     func (c *gobClientCodec) WriteRequest(r *Request, body inter
189         if err = c.enc.Encode(r); err != nil {
190             return
191         }
192         if err = c.enc.Encode(body); err != nil {

```

```

193         return
194     }
195     return c.encBuf.Flush()
196 }
197
198 func (c *gobClientCodec) ReadResponseHeader(r *Response) error {
199     return c.dec.Decode(r)
200 }
201
202 func (c *gobClientCodec) ReadResponseBody(body interface{}) {
203     return c.dec.Decode(body)
204 }
205
206 func (c *gobClientCodec) Close() error {
207     return c.rwc.Close()
208 }
209
210 // DialHTTP connects to an HTTP RPC server at the specified
211 // listening on the default HTTP RPC path.
212 func DialHTTP(network, address string) (*Client, error) {
213     return DialHTTPPath(network, address, DefaultRPCPath)
214 }
215
216 // DialHTTPPath connects to an HTTP RPC server
217 // at the specified network address and path.
218 func DialHTTPPath(network, address, path string) (*Client, error) {
219     var err error
220     conn, err := net.Dial(network, address)
221     if err != nil {
222         return nil, err
223     }
224     io.WriteString(conn, "CONNECT "+path+" HTTP/1.0\n\n")
225
226     // Require successful HTTP response
227     // before switching to RPC protocol.
228     resp, err := http.ReadResponse(bufio.NewReader(conn))
229     if err == nil && resp.Status == http.StatusOK {
230         return NewClient(conn), nil
231     }
232     if err == nil {
233         err = errors.New("unexpected HTTP response: " + resp.Status)
234     }
235     conn.Close()
236     return nil, &net.OpError{
237         Op:    "dial-http",
238         Net:   network + " " + address,
239         Addr:  nil,
240         Err:   err,
241     }
242 }

```

```

243
244 // Dial connects to an RPC server at the specified network a
245 func Dial(network, address string) (*Client, error) {
246     conn, err := net.Dial(network, address)
247     if err != nil {
248         return nil, err
249     }
250     return NewClient(conn), nil
251 }
252
253 func (client *Client) Close() error {
254     client.mutex.Lock()
255     if client.shutdown || client.closing {
256         client.mutex.Unlock()
257         return ErrShutdown
258     }
259     client.closing = true
260     client.mutex.Unlock()
261     return client.codec.Close()
262 }
263
264 // Go invokes the function asynchronously. It returns the C
265 // the invocation. The done channel will signal when the ca
266 // the same Call object. If done is nil, Go will allocate a
267 // If non-nil, done must be buffered or Go will deliberately
268 func (client *Client) Go(serviceMethod string, args interfac
269     call := new(Call)
270     call.ServiceMethod = serviceMethod
271     call.Args = args
272     call.Reply = reply
273     if done == nil {
274         done = make(chan *Call, 10) // buffered.
275     } else {
276         // If caller passes done != nil, it must arr
277         // done has enough buffer for the number of
278         // RPCs that will be using that channel. If
279         // is totally unbuffered, it's best not to r
280         if cap(done) == 0 {
281             log.Panic("rpc: done channel is unbu
282         }
283     }
284     call.Done = done
285     client.send(call)
286     return call
287 }
288
289 // Call invokes the named function, waits for it to complete
290 func (client *Client) Call(serviceMethod string, args interf
291     call := <-client.Go(serviceMethod, args, reply, make

```

```
292         return call.Error
293     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/rpc/debug.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package rpc
6
7 /*
8     Some HTML presented at http://machine:port/debug/rpc
9     Lists services, their methods, and some statistics,
10 */
11
12 import (
13     "fmt"
14     "net/http"
15     "sort"
16     "text/template"
17 )
18
19 const debugText = `
20     <body>
21     <title>Services</title>
22     {{range .}}
23     <hr>
24     Service {{.Name}}
25     <hr>
26         <table>
27         <th align=center>Method</th><th align=center
28         {{range .Method}}
29             <tr>
30                 <td align=left font=fixed>{{.Name}}(
31                 <td align=center>{{.Type.NumCalls}}<
32                 </tr>
33             {{end}}
34         </table>
35     {{end}}
36     </body>
37     </html>`
38
39 var debug = template.Must(template.New("RPC debug").Parse(debugText))
40
41 type debugMethod struct {
42     Type *methodType
43     Name string
44 }
```

```

45
46 type methodArray []debugMethod
47
48 type debugService struct {
49     Service *service
50     Name    string
51     Method  methodArray
52 }
53
54 type serviceArray []debugService
55
56 func (s serviceArray) Len() int           { return len(s) }
57 func (s serviceArray) Less(i, j int) bool { return s[i].Name
58 func (s serviceArray) Swap(i, j int)      { s[i], s[j] = s[j]
59
60 func (m methodArray) Len() int           { return len(m) }
61 func (m methodArray) Less(i, j int) bool { return m[i].Name
62 func (m methodArray) Swap(i, j int)      { m[i], m[j] = m[j]
63
64 type debugHTTP struct {
65     *Server
66 }
67
68 // Runs at /debug/rpc
69 func (server debugHTTP) ServeHTTP(w http.ResponseWriter, req
70     // Build a sorted version of the data.
71     var services = make(serviceArray, len(server.service
72     i := 0
73     server.mu.Lock()
74     for sname, service := range server.serviceMap {
75         services[i] = debugService{service, sname, m
76         j := 0
77         for mname, method := range service.method {
78             services[i].Method[j] = debugMethod{
79                 j++
80         }
81         sort.Sort(services[i].Method)
82         i++
83     }
84     server.mu.Unlock()
85     sort.Sort(services)
86     err := debug.Execute(w, services)
87     if err != nil {
88         fmt.Fprintln(w, "rpc: error executing templa
89     }
90 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/rpc/server.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6     Package rpc provides access to the exported methods
7     network or other I/O connection. A server registers
8     as a service with the name of the type of the object
9     methods of the object will be accessible remotely.
10    objects (services) of different types but it is an e
11    objects of the same type.
12
13    Only methods that satisfy these criteria will be mad
14    other methods will be ignored:
15
16        - the method is exported.
17        - the method has two arguments, both exporte
18        - the method's second argument is a pointer.
19        - the method has return type error.
20
21    In effect, the method must look schematically like
22
23        func (t *T) MethodName(argType T1, replyType
24
25    where T, T1 and T2 can be marshaled by encoding/gob.
26    These requirements apply even if a different codec i
27    (In future, these requirements may soften for custom
28
29    The method's first argument represents the arguments
30    second argument represents the result parameters to
31    The method's return value, if non-nil, is passed bac
32    sees as if created by errors.New.
33
34    The server may handle requests on a single connectio
35    typically it will create a network listener and call
36    listener, HandleHTTP and http.Serve.
37
38    A client wishing to use the service establishes a co
39    NewClient on the connection. The convenience functi
40    both steps for a raw network connection (an HTTP con
41    Client object has two methods, Call and Go, that spe
42    call, a pointer containing the arguments, and a poin
43    parameters.
44
```

45 The Call method waits for the remote call to complet
46 launches the call asynchronously and signals complet
47 structure's Done channel.

48

49 Unless an explicit codec is set up, package encoding
50 transport the data.

51

52 Here is a simple example. A server wishes to export

53

```
54 package server
```

55

```
56 type Args struct {
```

```
57     A, B int
```

58 }

59

```
60 type Quotient struct {
```

```
61     Quo, Rem int
```

62 }

63

```
64 type Arith int
```

65

```
66 func (t *Arith) Multiply(args *Args, reply *
```

```
67     *reply = args.A * args.B
```

```
68     return nil
```

69 }

70

```
71 func (t *Arith) Divide(args *Args, quo *Quot
```

```
72     if args.B == 0 {
```

```
73         return errors.New("divide by
```

74 }

```
75     quo.Quo = args.A / args.B
```

```
76     quo.Rem = args.A % args.B
```

```
77     return nil
```

78 }

79

80 The server calls (for HTTP service):

81

```
82     arith := new(Arith)
```

```
83     rpc.Register(arith)
```

```
84     rpc.HandleHTTP()
```

```
85     l, e := net.Listen("tcp", ":1234")
```

```
86     if e != nil {
```

```
87         log.Fatal("listen error:", e)
```

88 }

```
89     go http.Serve(l, nil)
```

90

91 At this point, clients can see a service "Arith" wit
92 "Arith.Divide". To invoke one, a client first dials

93

```
94     client, err := rpc.DialHTTP("tcp", serverAdd
```

```

95         if err != nil {
96             log.Fatal("dialing:", err)
97         }
98
99     Then it can make a remote call:
100
101         // Synchronous call
102         args := &server.Args{7,8}
103         var reply int
104         err = client.Call("Arith.Multiply", args, &r
105         if err != nil {
106             log.Fatal("arith error:", err)
107         }
108         fmt.Printf("Arith: %d*d=%d", args.A, args.B
109
110     or
111
112         // Asynchronous call
113         quotient := new(Quotient)
114         divCall := client.Go("Arith.Divide", args, &
115         replyCall := <-divCall.Done // will be e
116         // check errors, print, etc.
117
118     A server implementation will often provide a simple,
119     client.
120 */
121 package rpc
122
123 import (
124     "bufio"
125     "encoding/gob"
126     "errors"
127     "io"
128     "log"
129     "net"
130     "net/http"
131     "reflect"
132     "strings"
133     "sync"
134     "unicode"
135     "unicode/utf8"
136 )
137
138 const (
139     // Defaults used by HandleHTTP
140     DefaultRPCPath = "/_goRPC_"
141     DefaultDebugPath = "/debug/rpc"
142 )
143

```

```

144 // Precompute the reflect type for error. Can't use error d
145 // because Typeof takes an empty interface value. This is a
146 var typeOfError = reflect.TypeOf((*error)(nil)).Elem()
147
148 type methodType struct {
149     sync.Mutex // protects counters
150     method     reflect.Method
151     ArgType    reflect.Type
152     ReplyType  reflect.Type
153     numCalls   uint
154 }
155
156 type service struct {
157     name     string           // name of service
158     rcvr     reflect.Value    // receiver of methods
159     typ      reflect.Type     // type of the receive
160     method  map[string]*methodType // registered methods
161 }
162
163 // Request is a header written before every RPC call. It is
164 // but documented here as an aid to debugging, such as when
165 // network traffic.
166 type Request struct {
167     ServiceMethod string // format: "Service.Method"
168     Seq           uint64 // sequence number chosen by
169     next         *Request // for free list in Server
170 }
171
172 // Response is a header written before every RPC return. It
173 // but documented here as an aid to debugging, such as when
174 // network traffic.
175 type Response struct {
176     ServiceMethod string // echoes that of the Request
177     Seq           uint64 // echoes that of the request
178     Error         string // error, if any.
179     next         *Response // for free list in Server
180 }
181
182 // Server represents an RPC Server.
183 type Server struct {
184     mu          sync.Mutex // protects the serviceMap
185     serviceMap map[string]*service
186     reqLock    sync.Mutex // protects freeReq
187     freeReq    *Request
188     respLock   sync.Mutex // protects freeResp
189     freeResp   *Response
190 }
191
192 // NewServer returns a new Server.

```

```

193 func NewServer() *Server {
194     return &Server{serviceMap: make(map[string]*service)}
195 }
196
197 // DefaultServer is the default instance of *Server.
198 var DefaultServer = NewServer()
199
200 // Is this an exported - upper case - name?
201 func isExported(name string) bool {
202     rune, _ := utf8.DecodeRuneInString(name)
203     return unicode.IsUpper(rune)
204 }
205
206 // Is this type exported or a builtin?
207 func isExportedOrBuiltinType(t reflect.Type) bool {
208     for t.Kind() == reflect.Ptr {
209         t = t.Elem()
210     }
211     // PkgPath will be non-empty even for an exported ty
212     // so we need to check the type name as well.
213     return isExported(t.Name()) || t.PkgPath() == ""
214 }
215
216 // Register publishes in the server the set of methods of th
217 // receiver value that satisfy the following conditions:
218 //     - exported method
219 //     - two arguments, both pointers to exported structs
220 //     - one return value, of type error
221 // It returns an error if the receiver is not an exported ty
222 // suitable methods.
223 // The client accesses each method using a string of the for
224 // where Type is the receiver's concrete type.
225 func (server *Server) Register(rcvr interface{}) error {
226     return server.register(rcvr, "", false)
227 }
228
229 // RegisterName is like Register but uses the provided name
230 // instead of the receiver's concrete type.
231 func (server *Server) RegisterName(name string, rcvr interfa
232     return server.register(rcvr, name, true)
233 }
234
235 func (server *Server) register(rcvr interface{}, name string
236     server.mu.Lock()
237     defer server.mu.Unlock()
238     if server.serviceMap == nil {
239         server.serviceMap = make(map[string]*service)
240     }
241     s := new(service)
242     s.typ = reflect.TypeOf(rcvr)

```

```

243     s.rcvr = reflect.ValueOf(rcvr)
244     sname := reflect.Indirect(s.rcvr).Type().Name()
245     if useName {
246         sname = name
247     }
248     if sname == "" {
249         log.Fatal("rpc: no service name for type", s)
250     }
251     if !isExported(sname) && !useName {
252         s := "rpc Register: type " + sname + " is no
253         log.Print(s)
254         return errors.New(s)
255     }
256     if _, present := server.serviceMap[sname]; present {
257         return errors.New("rpc: service already defi
258     }
259     s.name = sname
260     s.method = make(map[string]*methodType)
261
262     // Install the methods
263     for m := 0; m < s.typ.NumMethod(); m++ {
264         method := s.typ.Method(m)
265         mtype := method.Type
266         mname := method.Name
267         // Method must be exported.
268         if method.PkgPath != "" {
269             continue
270         }
271         // Method needs three ins: receiver, *args,
272         if mtype.NumIn() != 3 {
273             log.Println("method", mname, "has wr
274             continue
275         }
276         // First arg need not be a pointer.
277         argType := mtype.In(1)
278         if !isExportedOrBuiltinType(argType) {
279             log.Println(mname, "argument type no
280             continue
281         }
282         // Second arg must be a pointer.
283         replyType := mtype.In(2)
284         if replyType.Kind() != reflect.Ptr {
285             log.Println("method", mname, "reply
286             continue
287         }
288         // Reply type must be exported.
289         if !isExportedOrBuiltinType(replyType) {
290             log.Println("method", mname, "reply
291             continue

```

```

292     }
293     // Method needs one out.
294     if mtype.NumOut() != 1 {
295         log.Println("method", mname, "has wr
296             continue
297     }
298     // The return type of the method must be err
299     if returnType := mtype.Out(0); returnType !=
300         log.Println("method", mname, "return
301             continue
302     }
303     s.method[mname] = &methodType{method: method
304 }
305
306     if len(s.method) == 0 {
307         s := "rpc Register: type " + sname + " has n
308         log.Print(s)
309         return errors.New(s)
310     }
311     server.serviceMap[s.name] = s
312     return nil
313 }
314
315 // A value sent as a placeholder for the server's response v
316 // receives an invalid request. It is never decoded by the c
317 // contains an error when it is used.
318 var invalidRequest = struct{}{}
319
320 func (server *Server) sendResponse(sending *sync.Mutex, req
321     resp := server.getResponse()
322     // Encode the response header
323     resp.ServiceMethod = req.ServiceMethod
324     if errmsg != "" {
325         resp.Error = errmsg
326         reply = invalidRequest
327     }
328     resp.Seq = req.Seq
329     sending.Lock()
330     err := codec.WriteResponse(resp, reply)
331     if err != nil {
332         log.Println("rpc: writing response:", err)
333     }
334     sending.Unlock()
335     server.freeResponse(resp)
336 }
337
338 func (m *methodType) NumCalls() (n uint) {
339     m.Lock()
340     n = m.numCalls

```

```

341         m.Unlock()
342         return n
343     }
344
345     func (s *service) call(server *Server, sending *sync.Mutex,
346         mtype.Lock()
347         mtype.numCalls++
348         mtype.Unlock()
349         function := mtype.method.Func
350         // Invoke the method, providing a new value for the
351         returnValues := function.Call([]reflect.Value{s.rcvr
352         // The return value for the method is an error.
353         errInter := returnValues[0].Interface()
354         errmsg := ""
355         if errInter != nil {
356             errmsg = errInter.(error).Error()
357         }
358         server.sendResponse(sending, req, replyv.Interface())
359         server.freeRequest(req)
360     }
361
362     type gobServerCodec struct {
363         rwc      io.ReadWriteCloser
364         dec      *gob.Decoder
365         enc      *gob.Encoder
366         encBuf   *bufio.Writer
367     }
368
369     func (c *gobServerCodec) ReadRequestHeader(r *Request) error
370     return c.dec.Decode(r)
371 }
372
373 func (c *gobServerCodec) ReadRequestBody(body interface{}) e
374 return c.dec.Decode(body)
375 }
376
377 func (c *gobServerCodec) WriteResponse(r *Response, body int
378 if err = c.enc.Encode(r); err != nil {
379     return
380 }
381 if err = c.enc.Encode(body); err != nil {
382     return
383 }
384 return c.encBuf.Flush()
385 }
386
387 func (c *gobServerCodec) Close() error {
388     return c.rwc.Close()
389 }
390

```

```

391 // ServeConn runs the server on a single connection.
392 // ServeConn blocks, serving the connection until the client
393 // The caller typically invokes ServeConn in a go statement.
394 // ServeConn uses the gob wire format (see package gob) on t
395 // connection. To use an alternate codec, use ServeCodec.
396 func (server *Server) ServeConn(conn io.ReadWriteCloser) {
397     buf := bufio.NewWriter(conn)
398     srv := &gobServerCodec{conn, gob.NewDecoder(conn), g
399     server.ServeCodec(srv)
400 }
401
402 // ServeCodec is like ServeConn but uses the specified codec
403 // decode requests and encode responses.
404 func (server *Server) ServeCodec(codec ServerCodec) {
405     sending := new(sync.Mutex)
406     for {
407         service, mtype, req, argv, replyv, keepReadi
408         if err != nil {
409             if err != io.EOF {
410                 log.Println("rpc:", err)
411             }
412             if !keepReading {
413                 break
414             }
415             // send a response if we actually ma
416             if req != nil {
417                 server.sendResponse(sending,
418                 server.freeRequest(req)
419             }
420             continue
421         }
422         go service.call(server, sending, mtype, req,
423     }
424     codec.Close()
425 }
426
427 // ServeRequest is like ServeCodec but synchronously serves
428 // It does not close the codec upon completion.
429 func (server *Server) ServeRequest(codec ServerCodec) error
430     sending := new(sync.Mutex)
431     service, mtype, req, argv, replyv, keepReading, err
432     if err != nil {
433         if !keepReading {
434             return err
435         }
436         // send a response if we actually managed to
437         if req != nil {
438             server.sendResponse(sending, req, in
439             server.freeRequest(req)

```

```

440         }
441         return err
442     }
443     service.call(server, sending, mtype, req, argv, repl
444     return nil
445 }
446
447 func (server *Server) getRequest() *Request {
448     server.reqLock.Lock()
449     req := server.freeReq
450     if req == nil {
451         req = new(Request)
452     } else {
453         server.freeReq = req.next
454         *req = Request{}
455     }
456     server.reqLock.Unlock()
457     return req
458 }
459
460 func (server *Server) freeRequest(req *Request) {
461     server.reqLock.Lock()
462     req.next = server.freeReq
463     server.freeReq = req
464     server.reqLock.Unlock()
465 }
466
467 func (server *Server) getResponse() *Response {
468     server.respLock.Lock()
469     resp := server.freeResp
470     if resp == nil {
471         resp = new(Response)
472     } else {
473         server.freeResp = resp.next
474         *resp = Response{}
475     }
476     server.respLock.Unlock()
477     return resp
478 }
479
480 func (server *Server) freeResponse(resp *Response) {
481     server.respLock.Lock()
482     resp.next = server.freeResp
483     server.freeResp = resp
484     server.respLock.Unlock()
485 }
486
487 func (server *Server) readRequest(codec ServerCodec) (servic
488     service, mtype, req, keepReading, err = server.readR

```

```

489         if err != nil {
490             if !keepReading {
491                 return
492             }
493             // discard body
494             codec.ReadRequestBody(nil)
495             return
496         }
497
498         // Decode the argument value.
499         argIsValue := false // if true, need to indirect bef
500         if mtype.ArgType.Kind() == reflect.Ptr {
501             argv = reflect.New(mtype.ArgType.Elem())
502         } else {
503             argv = reflect.New(mtype.ArgType)
504             argIsValue = true
505         }
506         // argv guaranteed to be a pointer now.
507         if err = codec.ReadRequestBody(argv.Interface()); er
508             return
509         }
510         if argIsValue {
511             argv = argv.Elem()
512         }
513
514         replyv = reflect.New(mtype.ReplyType.Elem())
515         return
516     }
517
518 func (server *Server) readRequestHeader(codec ServerCodec) (
519     // Grab the request header.
520     req = server.getRequest()
521     err = codec.ReadRequestHeader(req)
522     if err != nil {
523         req = nil
524         if err == io.EOF || err == io.ErrUnexpectedE
525             return
526         }
527         err = errors.New("rpc: server cannot decode
528             return
529     }
530
531     // We read the header successfully. If we see an er
532     // we can still recover and move on to the next requ
533     keepReading = true
534
535     serviceMethod := strings.Split(req.ServiceMethod, ".
536     if len(serviceMethod) != 2 {
537         err = errors.New("rpc: service/method reques
538         return

```

```

539     }
540     // Look up the request.
541     server.mu.Lock()
542     service = server.serviceMap[serviceMethod[0]]
543     server.mu.Unlock()
544     if service == nil {
545         err = errors.New("rpc: can't find service "
546             return
547     }
548     mtype = service.method[serviceMethod[1]]
549     if mtype == nil {
550         err = errors.New("rpc: can't find method " +
551     }
552     return
553 }
554
555 // Accept accepts connections on the listener and serves req
556 // for each incoming connection. Accept blocks; the caller
557 // invokes it in a go statement.
558 func (server *Server) Accept(lis net.Listener) {
559     for {
560         conn, err := lis.Accept()
561         if err != nil {
562             log.Fatal("rpc.Serve: accept:", err.)
563         }
564         go server.ServeConn(conn)
565     }
566 }
567
568 // Register publishes the receiver's methods in the DefaultS
569 func Register(rcvr interface{}) error { return DefaultServer
570
571 // RegisterName is like Register but uses the provided name
572 // instead of the receiver's concrete type.
573 func RegisterName(name string, rcvr interface{}) error {
574     return DefaultServer.RegisterName(name, rcvr)
575 }
576
577 // A ServerCodec implements reading of RPC requests and writ
578 // RPC responses for the server side of an RPC session.
579 // The server calls ReadRequestHeader and ReadRequestBody in
580 // to read requests from the connection, and it calls WriteR
581 // write a response back. The server calls Close when finis
582 // connection. ReadRequestBody may be called with a nil
583 // argument to force the body of the request to be read and
584 type ServerCodec interface {
585     ReadRequestHeader(*Request) error
586     ReadRequestBody(interface{}) error
587     WriteResponse(*Response, interface{}) error

```

```

588
589         Close() error
590     }
591
592     // ServeConn runs the DefaultServer on a single connection.
593     // ServeConn blocks, serving the connection until the client
594     // The caller typically invokes ServeConn in a go statement.
595     // ServeConn uses the gob wire format (see package gob) on t
596     // connection. To use an alternate codec, use ServeCodec.
597     func ServeConn(conn io.ReadWriteCloser) {
598         DefaultServer.ServeConn(conn)
599     }
600
601     // ServeCodec is like ServeConn but uses the specified codec
602     // decode requests and encode responses.
603     func ServeCodec(codec ServerCodec) {
604         DefaultServer.ServeCodec(codec)
605     }
606
607     // ServeRequest is like ServeCodec but synchronously serves
608     // It does not close the codec upon completion.
609     func ServeRequest(codec ServerCodec) error {
610         return DefaultServer.ServeRequest(codec)
611     }
612
613     // Accept accepts connections on the listener and serves req
614     // to DefaultServer for each incoming connection.
615     // Accept blocks; the caller typically invokes it in a go st
616     func Accept(lis net.Listener) { DefaultServer.Accept(lis) }
617
618     // Can connect to RPC service using HTTP CONNECT to rpcPath.
619     var connected = "200 Connected to Go RPC"
620
621     // ServeHTTP implements an http.Handler that answers RPC req
622     func (server *Server) ServeHTTP(w http.ResponseWriter, req *
623         if req.Method != "CONNECT" {
624             w.Header().Set("Content-Type", "text/plain;
625             w.WriteHeader(http.StatusMethodNotAllowed)
626             io.WriteString(w, "405 must CONNECT\n")
627             return
628         }
629         conn, _, err := w.(http.Hijacker).Hijack()
630         if err != nil {
631             log.Print("rpc hijacking ", req.RemoteAddr,
632             return
633         }
634         io.WriteString(conn, "HTTP/1.0 "+connected+"\n\n")
635         server.ServeConn(conn)
636     }

```

```
637
638 // HandleHTTP registers an HTTP handler for RPC messages on
639 // and a debugging handler on debugPath.
640 // It is still necessary to invoke http.Serve(), typically i
641 func (server *Server) HandleHTTP(rpcPath, debugPath string)
642     http.Handle(rpcPath, server)
643     http.Handle(debugPath, debugHTTP{server})
644 }
645
646 // HandleHTTP registers an HTTP handler for RPC messages to
647 // on DefaultRPCPath and a debugging handler on DefaultDebug
648 // It is still necessary to invoke http.Serve(), typically i
649 func HandleHTTP() {
650     DefaultServer.HandleHTTP(DefaultRPCPath, DefaultDebu
651 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/rpc/jsonrpc/client.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package jsonrpc implements a JSON-RPC ClientCodec and Ser
6 // for the rpc package.
7 package jsonrpc
8
9 import (
10     "encoding/json"
11     "fmt"
12     "io"
13     "net"
14     "net/rpc"
15     "sync"
16 )
17
18 type clientCodec struct {
19     dec *json.Decoder // for reading JSON values
20     enc *json.Encoder // for writing JSON values
21     c   io.Closer
22
23     // temporary work space
24     req clientRequest
25     resp clientResponse
26
27     // JSON-RPC responses include the request id but not
28     // Package rpc expects both.
29     // We save the request method in pending when sendin
30     // and then look it up by request ID when filling ou
31     mutex sync.Mutex // protects pending
32     pending map[uint64]string // map request id to metho
33 }
34
35 // NewClientCodec returns a new rpc.ClientCodec using JSON-R
36 func NewClientCodec(conn io.ReadWriteCloser) rpc.ClientCodec
37     return &clientCodec{
38         dec:    json.NewDecoder(conn),
39         enc:    json.NewEncoder(conn),
40         c:      conn,
41         pending: make(map[uint64]string),
```

```

42     }
43 }
44
45 type clientRequest struct {
46     Method string        `json:"method"`
47     Params [1]interface{} `json:"params"`
48     Id      uint64          `json:"id"`
49 }
50
51 func (c *clientCodec) WriteRequest(r *rpc.Request, param int
52     c.mutex.Lock()
53     c.pending[r.Seq] = r.ServiceMethod
54     c.mutex.Unlock()
55     c.req.Method = r.ServiceMethod
56     c.req.Params[0] = param
57     c.req.Id = r.Seq
58     return c.enc.Encode(&c.req)
59 }
60
61 type clientResponse struct {
62     Id      uint64          `json:"id"`
63     Result *json.RawMessage `json:"result"`
64     Error  interface{}     `json:"error"`
65 }
66
67 func (r *clientResponse) reset() {
68     r.Id = 0
69     r.Result = nil
70     r.Error = nil
71 }
72
73 func (c *clientCodec) ReadResponseHeader(r *rpc.Response) er
74     c.resp.reset()
75     if err := c.dec.Decode(&c.resp); err != nil {
76         return err
77     }
78
79     c.mutex.Lock()
80     r.ServiceMethod = c.pending[c.resp.Id]
81     delete(c.pending, c.resp.Id)
82     c.mutex.Unlock()
83
84     r.Error = ""
85     r.Seq = c.resp.Id
86     if c.resp.Error != nil {
87         x, ok := c.resp.Error.(string)
88         if !ok {
89             return fmt.Errorf("invalid error %v"
90         }
91         if x == "" {

```

```

92             x = "unspecified error"
93         }
94         r.Error = x
95     }
96     return nil
97 }
98
99 func (c *clientCodec) ReadResponseBody(x interface{}) error
100     if x == nil {
101         return nil
102     }
103     return json.Unmarshal(*c.resp.Result, x)
104 }
105
106 func (c *clientCodec) Close() error {
107     return c.c.Close()
108 }
109
110 // NewClient returns a new rpc.Client to handle requests to
111 // set of services at the other end of the connection.
112 func NewClient(conn io.ReadWriteCloser) *rpc.Client {
113     return rpc.NewClientWithCodec(NewClientCodec(conn))
114 }
115
116 // Dial connects to a JSON-RPC server at the specified network
117 func Dial(network, address string) (*rpc.Client, error) {
118     conn, err := net.Dial(network, address)
119     if err != nil {
120         return nil, err
121     }
122     return NewClient(conn), err
123 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/rpc/jsonrpc/server.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package jsonrpc
6
7 import (
8     "encoding/json"
9     "errors"
10    "io"
11    "net/rpc"
12    "sync"
13 )
14
15 type serverCodec struct {
16     dec *json.Decoder // for reading JSON values
17     enc *json.Encoder // for writing JSON values
18     c    io.Closer
19
20     // temporary work space
21     req  serverRequest
22     resp serverResponse
23
24     // JSON-RPC clients can use arbitrary json values as
25     // Package rpc expects uint64 request IDs.
26     // We assign uint64 sequence numbers to incoming req
27     // but save the original request ID in the pending map
28     // When rpc responds, we use the sequence number in
29     // the response to find the original request ID.
30     mutex sync.Mutex // protects seq, pending
31     seq   uint64
32     pending map[uint64]*json.RawMessage
33 }
34
35 // NewServerCodec returns a new rpc.ServerCodec using JSON-R
36 func NewServerCodec(conn io.ReadWriteCloser) rpc.ServerCodec
37     return &serverCodec{
38         dec:    json.NewDecoder(conn),
39         enc:    json.NewEncoder(conn),
40         c:      conn,
41         pending: make(map[uint64]*json.RawMessage),
```

```

42     }
43 }
44
45 type serverRequest struct {
46     Method string          `json:"method"`
47     Params  *json.RawMessage `json:"params"`
48     Id      *json.RawMessage `json:"id"`
49 }
50
51 func (r *serverRequest) reset() {
52     r.Method = ""
53     if r.Params != nil {
54         *r.Params = (*r.Params)[0:0]
55     }
56     if r.Id != nil {
57         *r.Id = (*r.Id)[0:0]
58     }
59 }
60
61 type serverResponse struct {
62     Id      *json.RawMessage `json:"id"`
63     Result interface{}   `json:"result"`
64     Error  interface{}   `json:"error"`
65 }
66
67 func (c *serverCodec) ReadRequestHeader(r *rpc.Request) error {
68     c.req.reset()
69     if err := c.dec.Decode(&c.req); err != nil {
70         return err
71     }
72     r.ServiceMethod = c.req.Method
73
74     // JSON request id can be any JSON value;
75     // RPC package expects uint64. Translate to
76     // internal uint64 and save JSON on the side.
77     c.mutex.Lock()
78     c.seq++
79     c.pending[c.seq] = c.req.Id
80     c.req.Id = nil
81     r.Seq = c.seq
82     c.mutex.Unlock()
83
84     return nil
85 }
86
87 func (c *serverCodec) ReadRequestBody(x interface{}) error {
88     if x == nil {
89         return nil
90     }
91     // JSON params is array value.

```

```

92         // RPC params is struct.
93         // Unmarshal into array containing struct for now.
94         // Should think about making RPC more general.
95         var params [1]interface{}
96         params[0] = x
97         return json.Unmarshal(*c.req.Params, &params)
98     }
99
100     var null = json.RawMessage([]byte("null"))
101
102     func (c *serverCodec) WriteResponse(r *rpc.Response, x inter
103         var resp serverResponse
104         c.mutex.Lock()
105         b, ok := c.pending[r.Seq]
106         if !ok {
107             c.mutex.Unlock()
108             return errors.New("invalid sequence number i
109         }
110         delete(c.pending, r.Seq)
111         c.mutex.Unlock()
112
113         if b == nil {
114             // Invalid request so no id. Use JSON null.
115             b = &null
116         }
117         resp.Id = b
118         resp.Result = x
119         if r.Error == "" {
120             resp.Error = nil
121         } else {
122             resp.Error = r.Error
123         }
124         return c.enc.Encode(resp)
125     }
126
127     func (c *serverCodec) Close() error {
128         return c.c.Close()
129     }
130
131     // ServeConn runs the JSON-RPC server on a single connection
132     // ServeConn blocks, serving the connection until the client
133     // The caller typically invokes ServeConn in a go statement.
134     func ServeConn(conn io.ReadWriteCloser) {
135         rpc.ServeCodec(NewServerCodec(conn))
136     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/smtp/auth.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package smtp
6
7 import (
8     "crypto/hmac"
9     "crypto/md5"
10    "errors"
11    "fmt"
12 )
13
14 // Auth is implemented by an SMTP authentication mechanism.
15 type Auth interface {
16     // Start begins an authentication with a server.
17     // It returns the name of the authentication protocol
18     // and optionally data to include in the initial AUTH
19     // sent to the server. It can return proto == "" to
20     // that the authentication should be skipped.
21     // If it returns a non-nil error, the SMTP client aborts
22     // the authentication attempt and closes the connection.
23     Start(server *ServerInfo) (proto string, toServer []byte, error)
24
25     // Next continues the authentication. The server has
26     // the fromServer data. If more is true, the server
27     // response, which Next should return as toServer; otherwise,
28     // Next should return toServer == nil.
29     // If Next returns a non-nil error, the SMTP client aborts
30     // the authentication attempt and closes the connection.
31     Next(fromServer []byte, more bool) (toServer []byte, error)
32 }
33
34 // ServerInfo records information about an SMTP server.
35 type ServerInfo struct {
36     Name string // SMTP server name
37     TLS  bool   // using TLS, with valid certificate
38     Auth []string // advertised authentication mechanisms
39 }
40
41 type plainAuth struct {
42     identity, username, password string
43     host                          string
44 }
```

```

45
46 // PlainAuth returns an Auth that implements the PLAIN auth
47 // mechanism as defined in RFC 4616.
48 // The returned Auth uses the given username and password to
49 // on TLS connections to host and act as identity. Usually i
50 // left blank to act as username.
51 func PlainAuth(identity, username, password, host string) Au
52     return &plainAuth{identity, username, password, host
53 }
54
55 func (a *plainAuth) Start(server *ServerInfo) (string, []byt
56     if !server.TLS {
57         return "", nil, errors.New("unencrypted conn
58     }
59     if server.Name != a.host {
60         return "", nil, errors.New("wrong host name"
61     }
62     resp := []byte(a.identity + "\x00" + a.username + "\
63     return "PLAIN", resp, nil
64 }
65
66 func (a *plainAuth) Next(fromServer []byte, more bool) ([]by
67     if more {
68         // We've already sent everything.
69         return nil, errors.New("unexpected server ch
70     }
71     return nil, nil
72 }
73
74 type cramMD5Auth struct {
75     username, secret string
76 }
77
78 // CRAMMD5Auth returns an Auth that implements the CRAM-MD5
79 // mechanism as defined in RFC 2195.
80 // The returned Auth uses the given username and secret to a
81 // to the server using the challenge-response mechanism.
82 func CRAMMD5Auth(username, secret string) Auth {
83     return &cramMD5Auth{username, secret}
84 }
85
86 func (a *cramMD5Auth) Start(server *ServerInfo) (string, []b
87     return "CRAM-MD5", nil, nil
88 }
89
90 func (a *cramMD5Auth) Next(fromServer []byte, more bool) ([]
91     if more {
92         d := hmac.New(md5.New, []byte(a.secret))
93         d.Write(fromServer)
94         s := make([]byte, 0, d.Size())

```

```
95             return []byte(fmt.Sprintf("%s %x", a.usernam
96         }
97     return nil, nil
98 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/smtp/smtp.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package smtp implements the Simple Mail Transfer Protocol
6 // It also implements the following extensions:
7 //     8BITMIME RFC 1652
8 //     AUTH     RFC 2554
9 //     STARTTLS RFC 3207
10 // Additional extensions may be handled by clients.
11 package smtp
12
13 import (
14     "crypto/tls"
15     "encoding/base64"
16     "io"
17     "net"
18     "net/textproto"
19     "strings"
20 )
21
22 // A Client represents a client connection to an SMTP server
23 type Client struct {
24     // Text is the textproto.Conn used by the Client. It
25     // clients to add extensions.
26     Text *textproto.Conn
27     // keep a reference to the connection so it can be u
28     // connection later
29     conn net.Conn
30     // whether the Client is using TLS
31     tls bool
32     serverName string
33     // map of supported extensions
34     ext map[string]string
35     // supported auth mechanisms
36     auth []string
37 }
38
39 // Dial returns a new Client connected to an SMTP server at
40 func Dial(addr string) (*Client, error) {
41     conn, err := net.Dial("tcp", addr)
42     if err != nil {
43         return nil, err
44     }
45 }
```

```

45         host := addr[:strings.Index(addr, ":")]
46         return NewClient(conn, host)
47     }
48
49     // NewClient returns a new Client using an existing connecti
50     // server name to be used when authenticating.
51     func NewClient(conn net.Conn, host string) (*Client, error)
52         text := textproto.NewConn(conn)
53         _, _, err := text.ReadResponse(220)
54         if err != nil {
55             text.Close()
56             return nil, err
57         }
58         c := &Client{Text: text, conn: conn, serverName: hos
59         err = c.ehlo()
60         if err != nil {
61             err = c.helo()
62         }
63         return c, err
64     }
65
66     // cmd is a convenience function that sends a command and re
67     func (c *Client) cmd(expectCode int, format string, args ...
68         id, err := c.Text.Cmd(format, args...)
69         if err != nil {
70             return 0, "", err
71         }
72         c.Text.StartResponse(id)
73         defer c.Text.EndResponse(id)
74         code, msg, err := c.Text.ReadResponse(expectCode)
75         return code, msg, err
76     }
77
78     // helo sends the HELO greeting to the server. It should be
79     // server does not support ehlo.
80     func (c *Client) helo() error {
81         c.ext = nil
82         _, _, err := c.cmd(250, "HELO localhost")
83         return err
84     }
85
86     // ehlo sends the EHLO (extended hello) greeting to the serv
87     // should be the preferred greeting for servers that support
88     func (c *Client) ehlo() error {
89         _, msg, err := c.cmd(250, "EHLO localhost")
90         if err != nil {
91             return err
92         }
93         ext := make(map[string]string)
94         extList := strings.Split(msg, "\n")

```

```

95         if len(extList) > 1 {
96             extList = extList[1:]
97             for _, line := range extList {
98                 args := strings.SplitN(line, " ", 2)
99                 if len(args) > 1 {
100                     ext[args[0]] = args[1]
101                 } else {
102                     ext[args[0]] = ""
103                 }
104             }
105         }
106         if mechs, ok := ext["AUTH"]; ok {
107             c.auth = strings.Split(mechs, " ")
108         }
109         c.ext = ext
110         return err
111     }
112
113     // StartTLS sends the STARTTLS command and encrypts all furt
114     // Only servers that advertise the STARTTLS extension suppor
115     func (c *Client) StartTLS(config *tls.Config) error {
116         _, _, err := c.cmd(220, "STARTTLS")
117         if err != nil {
118             return err
119         }
120         c.conn = tls.Client(c.conn, config)
121         c.Text = textproto.NewConn(c.conn)
122         c.tls = true
123         return c.ehlo()
124     }
125
126     // Verify checks the validity of an email address on the ser
127     // If Verify returns nil, the address is valid. A non-nil re
128     // does not necessarily indicate an invalid address. Many se
129     // will not verify addresses for security reasons.
130     func (c *Client) Verify(addr string) error {
131         _, _, err := c.cmd(250, "VRFY %s", addr)
132         return err
133     }
134
135     // Auth authenticates a client using the provided authentica
136     // A failed authentication closes the connection.
137     // Only servers that advertise the AUTH extension support th
138     func (c *Client) Auth(a Auth) error {
139         encoding := base64.StdEncoding
140         mech, resp, err := a.Start(&ServerInfo{c.serverName,
141             if err != nil {
142                 c.Quit()
143                 return err

```

```

144     }
145     resp64 := make([]byte, encoding.EncodedLen(len(resp))
146     encoding.Encode(resp64, resp)
147     code, msg64, err := c.cmd(0, "AUTH %s %s", mech, res
148     for err == nil {
149         var msg []byte
150         switch code {
151             case 334:
152                 msg, err = encoding.DecodeString(msg
153             case 235:
154                 // the last message isn't base64 bec
155                 msg = []byte(msg64)
156             default:
157                 err = &textproto.Error{Code: code, M
158         }
159         resp, err = a.Next(msg, code == 334)
160         if err != nil {
161             // abort the AUTH
162             c.cmd(501, "")
163             c.Quit()
164             break
165         }
166         if resp == nil {
167             break
168         }
169         resp64 = make([]byte, encoding.EncodedLen(le
170         encoding.Encode(resp64, resp)
171         code, msg64, err = c.cmd(0, string(resp64))
172     }
173     return err
174 }
175
176 // Mail issues a MAIL command to the server using the provid
177 // If the server supports the 8BITMIME extension, Mail adds
178 // parameter.
179 // This initiates a mail transaction and is followed by one
180 func (c *Client) Mail(from string) error {
181     cmdStr := "MAIL FROM:<%s>"
182     if c.ext != nil {
183         if _, ok := c.ext["8BITMIME"]; ok {
184             cmdStr += " BODY=8BITMIME"
185         }
186     }
187     _, _, err := c.cmd(250, cmdStr, from)
188     return err
189 }
190
191 // Rcpt issues a RCPT command to the server using the provid
192 // A call to Rcpt must be preceded by a call to Mail and may

```

```

193 // a Data call or another Rcpt call.
194 func (c *Client) Rcpt(to string) error {
195     _, _, err := c.cmd(25, "RCPT TO:<%s>", to)
196     return err
197 }
198
199 type dataCloser struct {
200     c *Client
201     io.WriteCloser
202 }
203
204 func (d *dataCloser) Close() error {
205     d.WriteCloser.Close()
206     _, _, err := d.c.Text.ReadResponse(250)
207     return err
208 }
209
210 // Data issues a DATA command to the server and returns a wr
211 // can be used to write the data. The caller should close th
212 // before calling any more methods on c.
213 // A call to Data must be preceded by one or more calls to R
214 func (c *Client) Data() (io.WriteCloser, error) {
215     _, _, err := c.cmd(354, "DATA")
216     if err != nil {
217         return nil, err
218     }
219     return &dataCloser{c, c.Text.DotWriter()}, nil
220 }
221
222 // SendMail connects to the server at addr, switches to TLS
223 // authenticates with mechanism a if possible, and then send
224 // address from, to addresses to, with message msg.
225 func SendMail(addr string, a Auth, from string, to []string,
226     c, err := Dial(addr)
227     if err != nil {
228         return err
229     }
230     if ok, _ := c.Extension("STARTTLS"); ok {
231         if err = c.StartTLS(nil); err != nil {
232             return err
233         }
234     }
235     if a != nil && c.ext != nil {
236         if _, ok := c.ext["AUTH"]; ok {
237             if err = c.Auth(a); err != nil {
238                 return err
239             }
240         }
241     }
242     if err = c.Mail(from); err != nil {

```

```

243         return err
244     }
245     for _, addr := range to {
246         if err = c.Rcpt(addr); err != nil {
247             return err
248         }
249     }
250     w, err := c.Data()
251     if err != nil {
252         return err
253     }
254     _, err = w.Write(msg)
255     if err != nil {
256         return err
257     }
258     err = w.Close()
259     if err != nil {
260         return err
261     }
262     return c.Quit()
263 }
264
265 // Extension reports whether an extension is support by the
266 // The extension name is case-insensitive. If the extension
267 // Extension also returns a string that contains any paramet
268 // server specifies for the extension.
269 func (c *Client) Extension(ext string) (bool, string) {
270     if c.ext == nil {
271         return false, ""
272     }
273     ext = strings.ToUpper(ext)
274     param, ok := c.ext[ext]
275     return ok, param
276 }
277
278 // Reset sends the RSET command to the server, aborting the
279 // transaction.
280 func (c *Client) Reset() error {
281     _, _, err := c.cmd(250, "RSET")
282     return err
283 }
284
285 // Quit sends the QUIT command and closes the connection to
286 func (c *Client) Quit() error {
287     _, _, err := c.cmd(221, "QUIT")
288     if err != nil {
289         return err
290     }
291     return c.Text.Close()

```

292 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/textproto/header.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package textproto
6
7 // A MIMEHeader represents a MIME-style header mapping
8 // keys to sets of values.
9 type MIMEHeader map[string][]string
10
11 // Add adds the key, value pair to the header.
12 // It appends to any existing values associated with key.
13 func (h MIMEHeader) Add(key, value string) {
14     key = CanonicalMIMEHeaderKey(key)
15     h[key] = append(h[key], value)
16 }
17
18 // Set sets the header entries associated with key to
19 // the single element value. It replaces any existing
20 // values associated with key.
21 func (h MIMEHeader) Set(key, value string) {
22     h[CanonicalMIMEHeaderKey(key)] = []string{value}
23 }
24
25 // Get gets the first value associated with the given key.
26 // If there are no values associated with the key, Get returns
27 // an empty string. Get is a convenience method. For more complex queries,
28 // access the map directly.
29 func (h MIMEHeader) Get(key string) string {
30     if h == nil {
31         return ""
32     }
33     v := h[CanonicalMIMEHeaderKey(key)]
34     if len(v) == 0 {
35         return ""
36     }
37     return v[0]
38 }
39
40 // Del deletes the values associated with key.
41 func (h MIMEHeader) Del(key string) {
```

```
42         delete(h, CanonicalMIMEHeaderKey(key))
43     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/textproto/pipeline.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package textproto
6
7 import (
8     "sync"
9 )
10
11 // A Pipeline manages a pipelined in-order request/response
12 //
13 // To use a Pipeline p to manage multiple clients on a connection,
14 // each client should run:
15 //
16 //     id := p.Next() // take a number
17 //
18 //     p.StartRequest(id) // wait for turn to send request
19 //     «send request»
20 //     p.EndRequest(id) // notify Pipeline that request is done
21 //
22 //     p.StartResponse(id) // wait for turn to read response
23 //     «read response»
24 //     p.EndResponse(id) // notify Pipeline that response is done
25 //
26 // A pipelined server can use the same calls to ensure that
27 // responses computed in parallel are written in the correct order.
28 type Pipeline struct {
29     mu sync.Mutex
30     id  uint
31     request sequencer
32     response sequencer
33 }
34
35 // Next returns the next id for a request/response pair.
36 func (p *Pipeline) Next() uint {
37     p.mu.Lock()
38     id := p.id
39     p.id++
40     p.mu.Unlock()
41     return id
42 }
```

```

42 }
43
44 // StartRequest blocks until it is time to send (or, if this
45 // the request with the given id.
46 func (p *Pipeline) StartRequest(id uint) {
47     p.request.Start(id)
48 }
49
50 // EndRequest notifies p that the request with the given id
51 // (or, if this is a server, received).
52 func (p *Pipeline) EndRequest(id uint) {
53     p.request.End(id)
54 }
55
56 // StartResponse blocks until it is time to receive (or, if
57 // the request with the given id.
58 func (p *Pipeline) StartResponse(id uint) {
59     p.response.Start(id)
60 }
61
62 // EndResponse notifies p that the response with the given i
63 // (or, if this is a server, sent).
64 func (p *Pipeline) EndResponse(id uint) {
65     p.response.End(id)
66 }
67
68 // A sequencer schedules a sequence of numbered events that
69 // happen in order, one after the other. The event numberin
70 // at 0 and increment without skipping. The event number wr
71 // safely as long as there are not 2^32 simultaneous events
72 type sequencer struct {
73     mu    sync.Mutex
74     id    uint
75     wait map[uint]chan uint
76 }
77
78 // Start waits until it is time for the event numbered id to
79 // That is, except for the first event, it waits until End(i
80 // been called.
81 func (s *sequencer) Start(id uint) {
82     s.mu.Lock()
83     if s.id == id {
84         s.mu.Unlock()
85         return
86     }
87     c := make(chan uint)
88     if s.wait == nil {
89         s.wait = make(map[uint]chan uint)
90     }
91     s.wait[id] = c

```

```

92         s.mu.Unlock()
93         <-c
94     }
95
96     // End notifies the sequencer that the event numbered id has
97     // allowing it to schedule the event numbered id+1. It is a
98     // to call End with an id that is not the number of the acti
99     func (s *sequencer) End(id uint) {
100         s.mu.Lock()
101         if s.id != id {
102             panic("out of sync")
103         }
104         id++
105         s.id = id
106         if s.wait == nil {
107             s.wait = make(map[uint]chan uint)
108         }
109         c, ok := s.wait[id]
110         if ok {
111             delete(s.wait, id)
112         }
113         s.mu.Unlock()
114         if ok {
115             c <- 1
116         }
117     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/textproto/reader.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package textproto
6
7 import (
8     "bufio"
9     "bytes"
10    "io"
11    "io/ioutil"
12    "strconv"
13    "strings"
14 )
15
16 // BUG(rsc): To let callers manage exposure to denial of ser
17 // attacks, Reader should allow them to set and reset a limi
18 // the number of bytes read from the connection.
19
20 // A Reader implements convenience methods for reading reque
21 // or responses from a text protocol network connection.
22 type Reader struct {
23     R    *bufio.Reader
24     dot *dotReader
25     buf []byte // a re-usable buffer for readContinuedLi
26 }
27
28 // NewReader returns a new Reader reading from r.
29 func NewReader(r *bufio.Reader) *Reader {
30     return &Reader{R: r}
31 }
32
33 // ReadLine reads a single line from r,
34 // eliding the final \n or \r\n from the returned string.
35 func (r *Reader) ReadLine() (string, error) {
36     line, err := r.readLineSlice()
37     return string(line), err
38 }
39
40 // ReadLineBytes is like ReadLine but returns a []byte inste
41 func (r *Reader) ReadLineBytes() ([]byte, error) {
```

```

42         line, err := r.readLineSlice()
43         if line != nil {
44             buf := make([]byte, len(line))
45             copy(buf, line)
46             line = buf
47         }
48         return line, err
49     }
50
51     func (r *Reader) readLineSlice() ([]byte, error) {
52         r.closeDot()
53         var line []byte
54         for {
55             l, more, err := r.R.ReadLine()
56             if err != nil {
57                 return nil, err
58             }
59             // Avoid the copy if the first call produced
60             if line == nil && !more {
61                 return l, nil
62             }
63             line = append(line, l...)
64             if !more {
65                 break
66             }
67         }
68         return line, nil
69     }
70
71     // ReadContinuedLine reads a possibly continued line from r,
72     // eliding the final trailing ASCII white space.
73     // Lines after the first are considered continuations if the
74     // begin with a space or tab character. In the returned dat
75     // continuation lines are separated from the previous line
76     // only by a single space: the newline and leading white spa
77     // are removed.
78     //
79     // For example, consider this input:
80     //
81     //     Line 1
82     //     continued...
83     //     Line 2
84     //
85     // The first call to ReadContinuedLine will return "Line 1 c
86     // and the second will return "Line 2".
87     //
88     // A line consisting of only white space is never continued.
89     //
90     func (r *Reader) ReadContinuedLine() (string, error) {
91         line, err := r.readContinuedLineSlice()

```

```

92         return string(line), err
93     }
94
95     // trim returns s with leading and trailing spaces and tabs
96     // It does not assume Unicode or UTF-8.
97     func trim(s []byte) []byte {
98         i := 0
99         for i < len(s) && (s[i] == ' ' || s[i] == '\t') {
100             i++
101         }
102         n := len(s)
103         for n > i && (s[n-1] == ' ' || s[n-1] == '\t') {
104             n--
105         }
106         return s[i:n]
107     }
108
109     // ReadContinuedLineBytes is like ReadContinuedLine but
110     // returns a []byte instead of a string.
111     func (r *Reader) ReadContinuedLineBytes() ([]byte, error) {
112         line, err := r.readContinuedLineSlice()
113         if line != nil {
114             buf := make([]byte, len(line))
115             copy(buf, line)
116             line = buf
117         }
118         return line, err
119     }
120
121     func (r *Reader) readContinuedLineSlice() ([]byte, error) {
122         // Read the first line.
123         line, err := r.readLineSlice()
124         if err != nil {
125             return nil, err
126         }
127         if len(line) == 0 { // blank line - no continuation
128             return line, nil
129         }
130
131         // ReadByte or the next readLineSlice will flush the
132         // copy the slice into buf.
133         r.buf = append(r.buf[:0], trim(line)...)
134
135         // Read continuation lines.
136         for r.skipSpace() > 0 {
137             line, err := r.readLineSlice()
138             if err != nil {
139                 break
140             }

```

```

141         r.buf = append(r.buf, ' ')
142         r.buf = append(r.buf, line...)
143     }
144     return r.buf, nil
145 }
146
147 // skipSpace skips R over all spaces and returns the number
148 func (r *Reader) skipSpace() int {
149     n := 0
150     for {
151         c, err := r.R.ReadByte()
152         if err != nil {
153             // bufio will keep err until next re
154             break
155         }
156         if c != ' ' && c != '\t' {
157             r.R.UnreadByte()
158             break
159         }
160         n++
161     }
162     return n
163 }
164
165 func (r *Reader) readCodeLine(expectCode int) (code int, con
166     line, err := r.ReadLine()
167     if err != nil {
168         return
169     }
170     return parseCodeLine(line, expectCode)
171 }
172
173 func parseCodeLine(line string, expectCode int) (code int, c
174     if len(line) < 4 || line[3] != ' ' && line[3] != '-'
175         err = ProtocolError("short response: " + lin
176         return
177     }
178     continued = line[3] == '-'
179     code, err = strconv.Atoi(line[0:3])
180     if err != nil || code < 100 {
181         err = ProtocolError("invalid response code:
182         return
183     }
184     message = line[4:]
185     if 1 <= expectCode && expectCode < 10 && code/100 !=
186         10 <= expectCode && expectCode < 100 && code
187         100 <= expectCode && expectCode < 1000 && co
188         err = &Error{code, message}
189     }

```

```

190         return
191     }
192
193 // ReadCodeLine reads a response code line of the form
194 //     code message
195 // where code is a 3-digit status code and the message
196 // extends to the rest of the line.  An example of such a li
197 //     220 plan9.bell-labs.com ESMTP
198 //
199 // If the prefix of the status does not match the digits in
200 // ReadCodeLine returns with err set to &Error{code, message
201 // For example, if expectCode is 31, an error will be return
202 // the status is not in the range [310,319].
203 //
204 // If the response is multi-line, ReadCodeLine returns an er
205 //
206 // An expectCode <= 0 disables the check of the status code.
207 //
208 func (r *Reader) ReadCodeLine(expectCode int) (code int, mes
209     code, continued, message, err := r.readCodeLine(expe
210     if err == nil && continued {
211         err = ProtocolError("unexpected multi-line r
212     }
213     return
214 }
215
216 // ReadResponse reads a multi-line response of the form:
217 //
218 //     code-message line 1
219 //     code-message line 2
220 //     ...
221 //     code message line n
222 //
223 // where code is a 3-digit status code. The first line start
224 // code and a hyphen. The response is terminated by a line t
225 // with the same code followed by a space. Each line in mess
226 // separated by a newline (\n).
227 //
228 // See page 36 of RFC 959 (http://www.ietf.org/rfc/rfc959.tx
229 // details.
230 //
231 // If the prefix of the status does not match the digits in
232 // ReadResponse returns with err set to &Error{code, message
233 // For example, if expectCode is 31, an error will be return
234 // the status is not in the range [310,319].
235 //
236 // An expectCode <= 0 disables the check of the status code.
237 //
238 func (r *Reader) ReadResponse(expectCode int) (code int, mes
239     code, continued, message, err := r.readCodeLine(expe

```

```

240         for err == nil && continued {
241             line, err := r.ReadLine()
242             if err != nil {
243                 return 0, "", err
244             }
245
246             var code2 int
247             var moreMessage string
248             code2, continued, moreMessage, err = parseCo
249             if err != nil || code2 != code {
250                 message += "\n" + strings.TrimRight(
251                     continued = true
252                     continue
253             }
254             message += "\n" + moreMessage
255         }
256         return
257     }
258
259     // DotReader returns a new Reader that satisfies Reads using
260     // decoded text of a dot-encoded block read from r.
261     // The returned Reader is only valid until the next call
262     // to a method on r.
263     //
264     // Dot encoding is a common framing used for data blocks
265     // in text protocols such as SMTP. The data consists of a s
266     // of lines, each of which ends in "\r\n". The sequence its
267     // ends at a line containing just a dot: ".\r\n". Lines beg
268     // with a dot are escaped with an additional dot to avoid
269     // looking like the end of the sequence.
270     //
271     // The decoded form returned by the Reader's Read method
272     // rewrites the "\r\n" line endings into the simpler "\n",
273     // removes leading dot escapes if present, and stops with er
274     // after consuming (and discarding) the end-of-sequence line
275     func (r *Reader) DotReader() io.Reader {
276         r.closeDot()
277         r.dot = &dotReader{r: r}
278         return r.dot
279     }
280
281     type dotReader struct {
282         r      *Reader
283         state int
284     }
285
286     // Read satisfies reads by decoding dot-encoded data read fr
287     func (d *dotReader) Read(b []byte) (n int, err error) {
288         // Run data through a simple state machine to

```

```

289 // elide leading dots, rewrite trailing \r\n into \n
290 // and detect ending .\r\n line.
291 const (
292     stateBeginLine = iota // beginning of line;
293     stateDot          // read . at beginning
294     stateDotCR       // read .\r at beginning
295     stateCR          // read \r (possibly a
296     stateData        // reading data in mid
297     stateEOF         // reached .\r\n end m
298 )
299 br := d.r.R
300 for n < len(b) && d.state != stateEOF {
301     var c byte
302     c, err = br.ReadByte()
303     if err != nil {
304         if err == io.EOF {
305             err = io.ErrUnexpectedEOF
306         }
307         break
308     }
309     switch d.state {
310     case stateBeginLine:
311         if c == '.' {
312             d.state = stateDot
313             continue
314         }
315         if c == '\r' {
316             d.state = stateCR
317             continue
318         }
319         d.state = stateData
320
321     case stateDot:
322         if c == '\r' {
323             d.state = stateDotCR
324             continue
325         }
326         if c == '\n' {
327             d.state = stateEOF
328             continue
329         }
330         d.state = stateData
331
332     case stateDotCR:
333         if c == '\n' {
334             d.state = stateEOF
335             continue
336         }
337         // Not part of .\r\n.

```

```

338         // Consume leading dot and emit save
339         br.UnreadByte()
340         c = '\r'
341         d.state = stateData
342
343     case stateCR:
344         if c == '\n' {
345             d.state = stateBeginLine
346             break
347         }
348         // Not part of \r\n.  Emit saved \r
349         br.UnreadByte()
350         c = '\r'
351         d.state = stateData
352
353     case stateData:
354         if c == '\r' {
355             d.state = stateCR
356             continue
357         }
358         if c == '\n' {
359             d.state = stateBeginLine
360         }
361     }
362     b[n] = c
363     n++
364 }
365 if err == nil && d.state == stateEOF {
366     err = io.EOF
367 }
368 if err != nil && d.r.dot == d {
369     d.r.dot = nil
370 }
371 return
372 }
373
374 // closeDot drains the current DotReader if any,
375 // making sure that it reads until the ending dot line.
376 func (r *Reader) closeDot() {
377     if r.dot == nil {
378         return
379     }
380     buf := make([]byte, 128)
381     for r.dot != nil {
382         // When Read reaches EOF or an error,
383         // it will set r.dot == nil.
384         r.dot.Read(buf)
385     }
386 }
387

```

```

388 // ReadDotBytes reads a dot-encoding and returns the decoded
389 //
390 // See the documentation for the DotReader method for detail
391 func (r *Reader) ReadDotBytes() ([]byte, error) {
392     return ioutil.ReadAll(r.DotReader())
393 }
394
395 // ReadDotLines reads a dot-encoding and returns a slice
396 // containing the decoded lines, with the final \r\n or \n e
397 //
398 // See the documentation for the DotReader method for detail
399 func (r *Reader) ReadDotLines() ([]string, error) {
400     // We could use ReadDotBytes and then Split it,
401     // but reading a line at a time avoids needing a
402     // large contiguous block of memory and is simpler.
403     var v []string
404     var err error
405     for {
406         var line string
407         line, err = r.ReadLine()
408         if err != nil {
409             if err == io.EOF {
410                 err = io.ErrUnexpectedEOF
411             }
412             break
413         }
414
415         // Dot by itself marks end; otherwise cut on
416         if len(line) > 0 && line[0] == '.' {
417             if len(line) == 1 {
418                 break
419             }
420             line = line[1:]
421         }
422         v = append(v, line)
423     }
424     return v, err
425 }
426
427 // ReadMIMEHeader reads a MIME-style header from r.
428 // The header is a sequence of possibly continued Key: Value
429 // ending in a blank line.
430 // The returned map m maps CanonicalMIMEHeaderKey(key) to a
431 // sequence of values in the same order encountered in the i
432 //
433 // For example, consider this input:
434 //
435 //     My-Key: Value 1
436 //     Long-Key: Even

```

```

437 //           Longer Value
438 //       My-Key: Value 2
439 //
440 // Given that input, ReadMIMEHeader returns the map:
441 //
442 //       map[string][]string{
443 //           "My-Key": {"Value 1", "Value 2"},
444 //           "Long-Key": {"Even Longer Value"},
445 //       }
446 //
447 func (r *Reader) ReadMIMEHeader() (MIMEHeader, error) {
448     m := make(MIMEHeader)
449     for {
450         kv, err := r.readContinuedLineSlice()
451         if len(kv) == 0 {
452             return m, err
453         }
454
455         // Key ends at first colon; must not have sp
456         i := bytes.IndexByte(kv, ':')
457         if i < 0 {
458             return m, ProtocolError("malformed M
459         }
460         key := string(kv[0:i])
461         if strings.Index(key, " ") >= 0 {
462             key = strings.TrimRight(key, " ")
463         }
464         key = CanonicalMIMEHeaderKey(key)
465
466         // Skip initial spaces in value.
467         i++ // skip colon
468         for i < len(kv) && (kv[i] == ' ' || kv[i] ==
469             i++)
470         }
471         value := string(kv[i:])
472
473         m[key] = append(m[key], value)
474
475         if err != nil {
476             return m, err
477         }
478     }
479     panic("unreachable")
480 }
481
482 // CanonicalMIMEHeaderKey returns the canonical format of th
483 // MIME header key s. The canonicalization converts the fir
484 // letter and any letter following a hyphen to upper case;
485 // the rest are converted to lowercase. For example, the

```

```

486 // canonical key for "accept-encoding" is "Accept-Encoding".
487 func CanonicalMIMEHeaderKey(s string) string {
488     // Quick check for canonical encoding.
489     needUpper := true
490     for i := 0; i < len(s); i++ {
491         c := s[i]
492         if needUpper && 'a' <= c && c <= 'z' {
493             goto MustRewrite
494         }
495         if !needUpper && 'A' <= c && c <= 'Z' {
496             goto MustRewrite
497         }
498         needUpper = c == '-'
499     }
500     return s
501
502 MustRewrite:
503     // Canonicalize: first letter upper case
504     // and upper case after each dash.
505     // (Host, User-Agent, If-Modified-Since).
506     // MIME headers are ASCII only, so no Unicode issues
507     a := []byte(s)
508     upper := true
509     for i, v := range a {
510         if v == '-' {
511             a[i] = '-'
512             upper = true
513             continue
514         }
515         if upper && 'a' <= v && v <= 'z' {
516             a[i] = v + 'A' - 'a'
517         }
518         if !upper && 'A' <= v && v <= 'Z' {
519             a[i] = v + 'a' - 'A'
520         }
521         upper = v == '-'
522     }
523     return string(a)
524 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/textproto/textproto.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package textproto implements generic support for text-based
6 // protocols in the style of HTTP, NNTP, and SMTP.
7 //
8 // The package provides:
9 //
10 // Error, which represents a numeric error response from
11 // a server.
12 //
13 // Pipeline, to manage pipelined requests and responses
14 // in a client.
15 //
16 // Reader, to read numeric response code lines,
17 // key: value headers, lines wrapped with leading spaces
18 // on continuation lines, and whole text blocks ending
19 // with a dot on a line by itself.
20 //
21 // Writer, to write dot-encoded text blocks.
22 //
23 // Conn, a convenient packaging of Reader, Writer, and Pipeline
24 // with a single network connection.
25 //
26 package textproto
27
28 import (
29     "bufio"
30     "fmt"
31     "io"
32     "net"
33 )
34
35 // An Error represents a numeric error response from a server.
36 type Error struct {
37     Code int
38     Msg  string
39 }
40
41 func (e *Error) Error() string {
```

```

42         return fmt.Sprintf("%03d %s", e.Code, e.Msg)
43     }
44
45     // A ProtocolError describes a protocol violation such
46     // as an invalid response or a hung-up connection.
47     type ProtocolError string
48
49     func (p ProtocolError) Error() string {
50         return string(p)
51     }
52
53     // A Conn represents a textual network protocol connection.
54     // It consists of a Reader and Writer to manage I/O
55     // and a Pipeline to sequence concurrent requests on the con
56     // These embedded types carry methods with them;
57     // see the documentation of those types for details.
58     type Conn struct {
59         Reader
60         Writer
61         Pipeline
62         conn io.ReadWriteCloser
63     }
64
65     // NewConn returns a new Conn using conn for I/O.
66     func NewConn(conn io.ReadWriteCloser) *Conn {
67         return &Conn{
68             Reader: Reader{R: bufio.NewReader(conn)},
69             Writer: Writer{W: bufio.NewWriter(conn)},
70             conn:   conn,
71         }
72     }
73
74     // Close closes the connection.
75     func (c *Conn) Close() error {
76         return c.conn.Close()
77     }
78
79     // Dial connects to the given address on the given network u
80     // and then returns a new Conn for the connection.
81     func Dial(network, addr string) (*Conn, error) {
82         c, err := net.Dial(network, addr)
83         if err != nil {
84             return nil, err
85         }
86         return NewConn(c), nil
87     }
88
89     // Cmd is a convenience method that sends a command after
90     // waiting its turn in the pipeline. The command text is th
91     // result of formatting format with args and appending \r\n.

```

```

92 // Cmd returns the id of the command, for use with StartResp
93 //
94 // For example, a client might run a HELP command that retur
95 // by using:
96 //
97 //     id, err := c.Cmd("HELP")
98 //     if err != nil {
99 //         return nil, err
100 //     }
101 //
102 //     c.StartResponse(id)
103 //     defer c.EndResponse(id)
104 //
105 //     if _, _, err = c.ReadCodeLine(110); err != nil {
106 //         return nil, err
107 //     }
108 //     text, err := c.ReadDotAll()
109 //     if err != nil {
110 //         return nil, err
111 //     }
112 //     return c.ReadCodeLine(250)
113 //
114 func (c *Conn) Cmd(format string, args ...interface{}) (id u
115     id = c.Next()
116     c.StartRequest(id)
117     err = c.PrintfLine(format, args...)
118     c.EndRequest(id)
119     if err != nil {
120         return 0, err
121     }
122     return id, nil
123 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/net/textproto/writer.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package textproto
6
7 import (
8     "bufio"
9     "fmt"
10    "io"
11 )
12
13 // A Writer implements convenience methods for writing
14 // requests or responses to a text protocol network connecti
15 type Writer struct {
16     W    *bufio.Writer
17     dot  *dotWriter
18 }
19
20 // NewWriter returns a new Writer writing to w.
21 func NewWriter(w *bufio.Writer) *Writer {
22     return &Writer{W: w}
23 }
24
25 var crnl = []byte{'\r', '\n'}
26 var dotcrnl = []byte{'.', '\r', '\n'}
27
28 // PrintfLine writes the formatted output followed by \r\n.
29 func (w *Writer) PrintfLine(format string, args ...interface)
30     w.closeDot()
31     fmt.Fprintf(w.W, format, args...)
32     w.W.Write(crnl)
33     return w.W.Flush()
34 }
35
36 // DotWriter returns a writer that can be used to write a do
37 // It takes care of inserting leading dots when necessary,
38 // translating line-ending \n into \r\n, and adding the fina
39 // when the DotWriter is closed. The caller should close th
40 // DotWriter before the next call to a method on w.
41 //
```

```

42 // See the documentation for Reader's DotReader method for d
43 func (w *Writer) DotWriter() io.WriterCloser {
44     w.closeDot()
45     w.dot = &dotWriter{w: w}
46     return w.dot
47 }
48
49 func (w *Writer) closeDot() {
50     if w.dot != nil {
51         w.dot.Close() // sets w.dot = nil
52     }
53 }
54
55 type dotWriter struct {
56     w      *Writer
57     state int
58 }
59
60 const (
61     wstateBeginLine = iota // beginning of line; initial
62     wstateCR             // wrote \r (possibly at end
63     wstateData          // writing data in middle of
64 )
65
66 func (d *dotWriter) Write(b []byte) (n int, err error) {
67     bw := d.w.W
68     for n < len(b) {
69         c := b[n]
70         switch d.state {
71         case wstateBeginLine:
72             d.state = wstateData
73             if c == '.' {
74                 // escape leading dot
75                 bw.WriteByte('.')
76             }
77             fallthrough
78
79         case wstateData:
80             if c == '\r' {
81                 d.state = wstateCR
82             }
83             if c == '\n' {
84                 bw.WriteByte('\r')
85                 d.state = wstateBeginLine
86             }
87
88         case wstateCR:
89             d.state = wstateData
90             if c == '\n' {
91                 d.state = wstateBeginLine

```

```

92             }
93         }
94         if err = bw.WriteByte(c); err != nil {
95             break
96         }
97         n++
98     }
99     return
100 }
101
102 func (d *dotWriter) Close() error {
103     if d.w.dot == d {
104         d.w.dot = nil
105     }
106     bw := d.w.W
107     switch d.state {
108     default:
109         bw.WriteByte('\r')
110         fallthrough
111     case wstateCR:
112         bw.WriteByte('\n')
113         fallthrough
114     case wstateBeginLine:
115         bw.Write(dotcrnl)
116     }
117     return bw.Flush()
118 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/net/url/url.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package url parses URLs and implements query escaping.
6 // See RFC 3986.
7 package url
8
9 import (
10     "errors"
11     "strconv"
12     "strings"
13 )
14
15 // Error reports an error and the operation and URL that cau
16 type Error struct {
17     Op string
18     URL string
19     Err error
20 }
21
22 func (e *Error) Error() string { return e.Op + " " + e.URL +
23
24 func ishex(c byte) bool {
25     switch {
26     case '0' <= c && c <= '9':
27         return true
28     case 'a' <= c && c <= 'f':
29         return true
30     case 'A' <= c && c <= 'F':
31         return true
32     }
33     return false
34 }
35
36 func unhex(c byte) byte {
37     switch {
38     case '0' <= c && c <= '9':
39         return c - '0'
40     case 'a' <= c && c <= 'f':
41         return c - 'a' + 10
42     case 'A' <= c && c <= 'F':
43         return c - 'A' + 10
44     }
```

```

45         return 0
46     }
47
48     type encoding int
49
50     const (
51         encodePath encoding = 1 + iota
52         encodeUserPassword
53         encodeQueryComponent
54         encodeFragment
55     )
56
57     type EscapeError string
58
59     func (e EscapeError) Error() string {
60         return "invalid URL escape " + strconv.Quote(string(
61     })
62
63     // Return true if the specified character should be escaped
64     // appearing in a URL string, according to RFC 3986.
65     // When 'all' is true the full range of reserved characters
66     func shouldEscape(c byte, mode encoding) bool {
67         // §2.3 Unreserved characters (alphanum)
68         if 'A' <= c && c <= 'Z' || 'a' <= c && c <= 'z' || '
69             return false
70     }
71
72     switch c {
73     case '-', '_', '.', '~': // §2.3 Unreserved characte
74         return false
75
76     case '$', '&', '+', ',', '/', ':', ';', '=', '?', '@
77         // Different sections of the URL allow a few
78         // the reserved characters to appear unescap
79         switch mode {
80         case encodePath: // §3.3
81             // The RFC allows : @ & = + $ but sa
82             // meaning to individual path segmen
83             // only manipulates the path as a wh
84             // last two as well. That leaves onl
85             return c == '?'
86
87         case encodeUserPassword: // §3.2.2
88             // The RFC allows ; : & = + $ , in u
89             // The parsing of userinfo treats :
90             return c == '@' || c == '/' || c ==
91
92         case encodeQueryComponent: // §3.4
93             // The RFC reserves (so we must esca
94             return true

```

```

95
96         case encodeFragment: // §4.1
97             // The RFC text is silent but the gr
98             // everything, so escape nothing.
99             return false
100         }
101     }
102
103     // Everything else must be escaped.
104     return true
105 }
106
107 // QueryUnescape does the inverse transformation of QueryEsc
108 // %AB into the byte 0xAB and '+' into ' ' (space). It retur
109 // any % is not followed by two hexadecimal digits.
110 func QueryUnescape(s string) (string, error) {
111     return unescape(s, encodeQueryComponent)
112 }
113
114 // unescape unescapes a string; the mode specifies
115 // which section of the URL string is being unescaped.
116 func unescape(s string, mode encoding) (string, error) {
117     // Count %, check that they're well-formed.
118     n := 0
119     hasPlus := false
120     for i := 0; i < len(s); {
121         switch s[i] {
122             case '%':
123                 n++
124                 if i+2 >= len(s) || !ishex(s[i+1]) |
125                     s = s[i:]
126                     if len(s) > 3 {
127                         s = s[0:3]
128                     }
129                     return "", EscapeError(s)
130                 }
131                 i += 3
132             case '+':
133                 hasPlus = mode == encodeQueryCompone
134                 i++
135             default:
136                 i++
137         }
138     }
139
140     if n == 0 && !hasPlus {
141         return s, nil
142     }
143

```

```

144     t := make([]byte, len(s)-2*n)
145     j := 0
146     for i := 0; i < len(s); {
147         switch s[i] {
148             case '%':
149                 t[j] = unhex(s[i+1])<<4 | unhex(s[i+
150                     j++
151                     i += 3
152             case '+':
153                 if mode == encodeQueryComponent {
154                     t[j] = ' '
155                 } else {
156                     t[j] = '+'
157                 }
158                 j++
159                 i++
160             default:
161                 t[j] = s[i]
162                 j++
163                 i++
164         }
165     }
166     return string(t), nil
167 }
168
169 // QueryEscape escapes the string so it can be safely placed
170 // inside a URL query.
171 func QueryEscape(s string) string {
172     return escape(s, encodeQueryComponent)
173 }
174
175 func escape(s string, mode encoding) string {
176     spaceCount, hexCount := 0, 0
177     for i := 0; i < len(s); i++ {
178         c := s[i]
179         if shouldEscape(c, mode) {
180             if c == ' ' && mode == encodeQueryCo
181                 spaceCount++
182             } else {
183                 hexCount++
184             }
185         }
186     }
187
188     if spaceCount == 0 && hexCount == 0 {
189         return s
190     }
191
192     t := make([]byte, len(s)+2*hexCount)

```

```

193         j := 0
194         for i := 0; i < len(s); i++ {
195             switch c := s[i]; {
196                 case c == ' ' && mode == encodeQueryComponen
197                     t[j] = '+'
198                     j++
199                 case shouldEscape(c, mode):
200                     t[j] = '%'
201                     t[j+1] = "0123456789ABCDEF"[c>>4]
202                     t[j+2] = "0123456789ABCDEF"[c&15]
203                     j += 3
204                 default:
205                     t[j] = s[i]
206                     j++
207             }
208         }
209         return string(t)
210     }
211
212     // A URL represents a parsed URL (technically, a URI referen
213     // The general form represented is:
214     //
215     //     scheme://[userinfo@]host/path[?query][#fragment]
216     //
217     // URLs that do not start with a slash after the scheme are
218     //
219     //     scheme:opaque[?query][#fragment]
220     //
221     type URL struct {
222         Scheme string
223         Opaque  string // encoded opaque data
224         User   *Userinfo // username and password informat
225         Host   string
226         Path   string
227         RawQuery string // encoded query values, without '?'
228         Fragment string // fragment for references, without
229     }
230
231     // User returns a Userinfo containing the provided username
232     // and no password set.
233     func User(username string) *Userinfo {
234         return &Userinfo{username, "", false}
235     }
236
237     // UserPassword returns a Userinfo containing the provided u
238     // and password.
239     // This functionality should only be used with legacy web si
240     // RFC 2396 warns that interpreting Userinfo this way
241     // ``is NOT RECOMMENDED, because the passing of authenticati
242     // information in clear text (such as URI) has proven to be

```

```

243 // security risk in almost every case where it has been used
244 func UserPassword(username, password string) *Userinfo {
245     return &Userinfo{username, password, true}
246 }
247
248 // The Userinfo type is an immutable encapsulation of userna
249 // password details for a URL. An existing Userinfo value is
250 // to have a username set (potentially empty, as allowed by
251 // and optionally a password.
252 type Userinfo struct {
253     username    string
254     password    string
255     passwordSet bool
256 }
257
258 // Username returns the username.
259 func (u *Userinfo) Username() string {
260     return u.username
261 }
262
263 // Password returns the password in case it is set, and whet
264 func (u *Userinfo) Password() (string, bool) {
265     if u.passwordSet {
266         return u.password, true
267     }
268     return "", false
269 }
270
271 // String returns the encoded userinfo information in the st
272 // of "username[:password]".
273 func (u *Userinfo) String() string {
274     s := escape(u.username, encodeUserPassword)
275     if u.passwordSet {
276         s += ":" + escape(u.password, encodeUserPass
277     }
278     return s
279 }
280
281 // Maybe rawurl is of the form scheme:path.
282 // (Scheme must be [a-zA-Z][a-zA-Z0-9+-.]*)
283 // If so, return scheme, path; else return "", rawurl.
284 func getscheme(rawurl string) (scheme, path string, err erro
285     for i := 0; i < len(rawurl); i++ {
286         c := rawurl[i]
287         switch {
288             case 'a' <= c && c <= 'z' || 'A' <= c && c <
289             // do nothing
290             case '0' <= c && c <= '9' || c == '+' || c =
291                 if i == 0 {

```

```

292         return "", rawurl, nil
293     }
294     case c == ':':
295         if i == 0 {
296             return "", "", errors.New("m
297         }
298         return rawurl[0:i], rawurl[i+1:], ni
299     default:
300         // we have encountered an invalid ch
301         // so there is no valid scheme
302         return "", rawurl, nil
303     }
304 }
305 return "", rawurl, nil
306 }
307
308 // Maybe s is of the form t c u.
309 // If so, return t, c u (or t, u if cutc == true).
310 // If not, return s, "".
311 func split(s string, c byte, cutc bool) (string, string) {
312     for i := 0; i < len(s); i++ {
313         if s[i] == c {
314             if cutc {
315                 return s[0:i], s[i+1:]
316             }
317             return s[0:i], s[i:]
318         }
319     }
320     return s, ""
321 }
322
323 // Parse parses rawurl into a URL structure.
324 // The rawurl may be relative or absolute.
325 func Parse(rawurl string) (url *URL, err error) {
326     // Cut off #frag
327     u, frag := split(rawurl, '#', true)
328     if url, err = parse(u, false); err != nil {
329         return nil, err
330     }
331     if frag == "" {
332         return url, nil
333     }
334     if url.Fragment, err = unescape(frag, encodeFragment
335         return nil, &Error{"parse", rawurl, err}
336     }
337     return url, nil
338 }
339
340 // ParseRequestURI parses rawurl into a URL structure. It a

```

```

341 // rawurl was received in an HTTP request, so the rawurl is
342 // only as an absolute URI or an absolute path.
343 // The string rawurl is assumed not to have a #fragment suff
344 // (Web browsers strip #fragment before sending the URL to a
345 func ParseRequestURI(rawurl string) (url *URL, err error) {
346     return parse(rawurl, true)
347 }
348
349 // parse parses a URL from a string in one of two contexts.
350 // viaRequest is true, the URL is assumed to have arrived vi
351 // in which case only absolute URLs or path-absolute relativ
352 // If viaRequest is false, all forms of relative URLs are al
353 func parse(rawurl string, viaRequest bool) (url *URL, err er
354     var rest string
355
356     if rawurl == "" {
357         err = errors.New("empty url")
358         goto Error
359     }
360     url = new(URL)
361
362     // Split off possible leading "http:", "mailto:", et
363     // Cannot contain escaped characters.
364     if url.Scheme, rest, err = getscheme(rawurl); err !=
365         goto Error
366     }
367
368     rest, url.RawQuery = split(rest, '?', true)
369
370     if !strings.HasPrefix(rest, "/") {
371         if url.Scheme != "" {
372             // We consider rootless paths per RF
373             url.Opaque = rest
374             return url, nil
375         }
376         if viaRequest {
377             err = errors.New("invalid URI for re
378             goto Error
379         }
380     }
381
382     if (url.Scheme != "" || !viaRequest) && strings.HasP
383     var authority string
384     authority, rest = split(rest[2:], '/', false
385     url.User, url.Host, err = parseAuthority(aut
386     if err != nil {
387         goto Error
388     }
389     if strings.Contains(url.Host, "%") {
390         err = errors.New("hexadecimal escape

```

```

391             goto Error
392         }
393     }
394     if url.Path, err = unescape(rest, encodePath); err != nil {
395         goto Error
396     }
397     return url, nil
398
399 Error:
400     return nil, &Error{"parse", rawurl, err}
401 }
402
403 func parseAuthority(authority string) (user *Userinfo, host string, err error) {
404     if strings.Index(authority, "@") < 0 {
405         host = authority
406         return
407     }
408     userinfo, host := split(authority, '@', true)
409     if strings.Index(userinfo, ":") < 0 {
410         if userinfo, err = unescape(userinfo, encodePath); err != nil {
411             return
412         }
413         user = User(userinfo)
414     } else {
415         username, password := split(userinfo, ':', true)
416         if username, err = unescape(username, encodePath); err != nil {
417             return
418         }
419         if password, err = unescape(password, encodePath); err != nil {
420             return
421         }
422         user = UserPassword(username, password)
423     }
424     return
425 }
426
427 // String reassembles the URL into a valid URL string.
428 func (u *URL) String() string {
429     // TODO: Rewrite to use bytes.Buffer
430     result := ""
431     if u.Scheme != "" {
432         result += u.Scheme + ":"
433     }
434     if u.Opaque != "" {
435         result += u.Opaque
436     } else {
437         if u.Host != "" || u.User != nil {
438             result += "://"
439             if u := u.User; u != nil {

```

```

440             result += u.String() + "@"
441         }
442         result += u.Host
443     }
444     result += escape(u.Path, encodePath)
445 }
446 if u.RawQuery != "" {
447     result += "?" + u.RawQuery
448 }
449 if u.Fragment != "" {
450     result += "#" + escape(u.Fragment, encodeFra
451 }
452 return result
453 }
454
455 // Values maps a string key to a list of values.
456 // It is typically used for query parameters and form values
457 // Unlike in the http.Header map, the keys in a Values map
458 // are case-sensitive.
459 type Values map[string][]string
460
461 // Get gets the first value associated with the given key.
462 // If there are no values associated with the key, Get retur
463 // the empty string. To access multiple values, use the map
464 // directly.
465 func (v Values) Get(key string) string {
466     if v == nil {
467         return ""
468     }
469     vs, ok := v[key]
470     if !ok || len(vs) == 0 {
471         return ""
472     }
473     return vs[0]
474 }
475
476 // Set sets the key to value. It replaces any existing
477 // values.
478 func (v Values) Set(key, value string) {
479     v[key] = []string{value}
480 }
481
482 // Add adds the key to value. It appends to any existing
483 // values associated with key.
484 func (v Values) Add(key, value string) {
485     v[key] = append(v[key], value)
486 }
487
488 // Del deletes the values associated with key.

```

```

489 func (v Values) Del(key string) {
490     delete(v, key)
491 }
492
493 // ParseQuery parses the URL-encoded query string and return
494 // a map listing the values specified for each key.
495 // ParseQuery always returns a non-nil map containing all th
496 // valid query parameters found; err describes the first dec
497 // encountered, if any.
498 func ParseQuery(query string) (m Values, err error) {
499     m = make(Values)
500     err = parseQuery(m, query)
501     return
502 }
503
504 func parseQuery(m Values, query string) (err error) {
505     for query != "" {
506         key := query
507         if i := strings.IndexAny(key, "&"); i >= 0 {
508             key, query = key[:i], key[i+1:]
509         } else {
510             query = ""
511         }
512         if key == "" {
513             continue
514         }
515         value := ""
516         if i := strings.Index(key, "="); i >= 0 {
517             key, value = key[:i], key[i+1:]
518         }
519         key, err1 := QueryUnescape(key)
520         if err1 != nil {
521             err = err1
522             continue
523         }
524         value, err1 = QueryUnescape(value)
525         if err1 != nil {
526             err = err1
527             continue
528         }
529         m[key] = append(m[key], value)
530     }
531     return err
532 }
533
534 // Encode encodes the values into ``URL encoded'' form.
535 // e.g. "foo=bar&bar=baz"
536 func (v Values) Encode() string {
537     if v == nil {
538         return ""

```

```

539     }
540     parts := make([]string, 0, len(v)) // will be large
541     for k, vs := range v {
542         prefix := QueryEscape(k) + "="
543         for _, v := range vs {
544             parts = append(parts, prefix+QueryEs
545         }
546     }
547     return strings.Join(parts, "&")
548 }
549
550 // resolvePath applies special path segments from refs and a
551 // them to base, per RFC 2396.
552 func resolvePath(basepath string, reffpath string) string {
553     base := strings.Split(basepath, "/")
554     refs := strings.Split(reffpath, "/")
555     if len(base) == 0 {
556         base = []string{""}
557     }
558     for idx, ref := range refs {
559         switch {
560         case ref == ".":
561             base[len(base)-1] = ""
562         case ref == "..":
563             newLen := len(base) - 1
564             if newLen < 1 {
565                 newLen = 1
566             }
567             base = base[0:newLen]
568             base[len(base)-1] = ""
569         default:
570             if idx == 0 || base[len(base)-1] ==
571                 base[len(base)-1] = ref
572             } else {
573                 base = append(base, ref)
574             }
575         }
576     }
577     return strings.Join(base, "/")
578 }
579
580 // IsAbs returns true if the URL is absolute.
581 func (u *URL) IsAbs() bool {
582     return u.Scheme != ""
583 }
584
585 // Parse parses a URL in the context of the receiver. The p
586 // may be relative or absolute. Parse returns nil, err on p
587 // failure, otherwise its return value is the same as Resolv

```

```

588 func (u *URL) Parse(ref string) (*URL, error) {
589     refurl, err := Parse(ref)
590     if err != nil {
591         return nil, err
592     }
593     return u.ResolveReference(refurl), nil
594 }
595
596 // ResolveReference resolves a URI reference to an absolute
597 // an absolute base URI, per RFC 2396 Section 5.2. The URI
598 // may be relative or absolute. ResolveReference always ret
599 // URL instance, even if the returned URL is identical to ei
600 // base or reference. If ref is an absolute URL, then Resolv
601 // ignores base and returns a copy of ref.
602 func (u *URL) ResolveReference(ref *URL) *URL {
603     if ref.IsAbs() {
604         url := *ref
605         return &url
606     }
607     // relativeURI = ( net_path | abs_path | rel_path )
608     url := *u
609     url.RawQuery = ref.RawQuery
610     url.Fragment = ref.Fragment
611     if ref.Opaque != "" {
612         url.Opaque = ref.Opaque
613         url.User = nil
614         url.Host = ""
615         url.Path = ""
616         return &url
617     }
618     if ref.Host != "" || ref.User != nil {
619         // The "net_path" case.
620         url.Host = ref.Host
621         url.User = ref.User
622     }
623     if strings.HasPrefix(ref.Path, "/") {
624         // The "abs_path" case.
625         url.Path = ref.Path
626     } else {
627         // The "rel_path" case.
628         path := resolvePath(u.Path, ref.Path)
629         if !strings.HasPrefix(path, "/") {
630             path = "/" + path
631         }
632         url.Path = path
633     }
634     return &url
635 }
636

```

```
637 // Query parses RawQuery and returns the corresponding value
638 func (u *URL) Query() Values {
639     v, _ := ParseQuery(u.RawQuery)
640     return v
641 }
642
643 // RequestURI returns the encoded path?query or opaque?query
644 // string that would be used in an HTTP request for u.
645 func (u *URL) RequestURI() string {
646     result := u.Opaque
647     if result == "" {
648         result = escape(u.Path, encodePath)
649         if result == "" {
650             result = "/"
651         }
652     }
653     if u.RawQuery != "" {
654         result += "?" + u.RawQuery
655     }
656     return result
657 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/dir_unix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package os
8
9 import (
10     "io"
11     "syscall"
12 )
13
14 const (
15     blockSize = 4096
16 )
17
18 func (f *File) readdirnames(n int) (names []string, err error) {
19     // If this file has no dirinfo, create one.
20     if f.dirinfo == nil {
21         f.dirinfo = new(dirInfo)
22         // The buffer must be at least a block long.
23         f.dirinfo.buf = make([]byte, blockSize)
24     }
25     d := f.dirinfo
26
27     size := n
28     if size <= 0 {
29         size = 100
30         n = -1
31     }
32
33     names = make([]string, 0, size) // Empty with room to
34     for n != 0 {
35         // Refill the buffer if necessary
36         if d.bufp >= d.nbuf {
37             d.bufp = 0
38             var errno error
39             d.nbuf, errno = syscall.ReadDirent(f)
40             if errno != nil {
41                 return names, NewSyscallError(errno, "syscall.ReadDirent")
42             }
43             if d.nbuf <= 0 {
44                 break // EOF
```

```
45         }
46     }
47
48     // Drain the buffer
49     var nb, nc int
50     nb, nc, names = syscall.ParseDirent(d.buf[d.
51     d.bufp += nb
52     n -= nc
53 }
54 if n >= 0 && len(names) == 0 {
55     return names, io.EOF
56 }
57 return names, nil
58 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/doc.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package os
6
7 import "time"
8
9 // FindProcess looks for a running process by its pid.
10 // The Process it returns can be used to obtain information
11 // about the underlying operating system process.
12 func FindProcess(pid int) (p *Process, err error) {
13     return findProcess(pid)
14 }
15
16 // StartProcess starts a new process with the program, argument
17 // specified by name, argv and attr.
18 //
19 // StartProcess is a low-level interface. The os/exec package
20 // higher-level interfaces.
21 //
22 // If there is an error, it will be of type *PathError.
23 func StartProcess(name string, argv []string, attr *ProcAttr)
24     return startProcess(name, argv, attr)
25 }
26
27 // Release releases any resources associated with the Process
28 // rendering it unusable in the future.
29 // Release only needs to be called if Wait is not.
30 func (p *Process) Release() error {
31     return p.release()
32 }
33
34 // Kill causes the Process to exit immediately.
35 func (p *Process) Kill() error {
36     return p.kill()
37 }
38
39 // Wait waits for the Process to exit, and then returns a
40 // ProcessState describing its status and an error, if any.
41 // Wait releases any resources associated with the Process.
42 func (p *Process) Wait() (*ProcessState, error) {
43     return p.wait()
44 }
```

```

45
46 // Signal sends a signal to the Process.
47 func (p *Process) Signal(sig Signal) error {
48     return p.signal(sig)
49 }
50
51 // UserTime returns the user CPU time of the exited process
52 func (p *ProcessState) UserTime() time.Duration {
53     return p.userTime()
54 }
55
56 // SystemTime returns the system CPU time of the exited proc
57 func (p *ProcessState) SystemTime() time.Duration {
58     return p.systemTime()
59 }
60
61 // Exited returns whether the program has exited.
62 func (p *ProcessState) Exited() bool {
63     return p.exited()
64 }
65
66 // Success reports whether the program exited successfully,
67 // such as with exit status 0 on Unix.
68 func (p *ProcessState) Success() bool {
69     return p.success()
70 }
71
72 // Sys returns system-dependent exit information about
73 // the process. Convert it to the appropriate underlying
74 // type, such as syscall.WaitStatus on Unix, to access its c
75 func (p *ProcessState) Sys() interface{} {
76     return p.sys()
77 }
78
79 // SysUsage returns system-dependent resource usage informat
80 // the exited process. Convert it to the appropriate underl
81 // type, such as *syscall.Rusage on Unix, to access its cont
82 func (p *ProcessState) SysUsage() interface{} {
83     return p.sysUsage()
84 }
85
86 // Hostname returns the host name reported by the kernel.
87 func Hostname() (name string, err error) {
88     return hostname()
89 }
90
91 // Readdir reads the contents of the directory associated wi
92 // returns an array of up to n FileInfo values, as would be
93 // by Lstat, in directory order. Subsequent calls on the sam
94 // further FileInfos.

```

```

95 //
96 // If n > 0, Readdir returns at most n FileInfo structures.
97 // Readdir returns an empty slice, it will return a non-nil
98 // explaining why. At the end of a directory, the error is i
99 //
100 // If n <= 0, Readdir returns all the FileInfo from the dire
101 // a single slice. In this case, if Readdir succeeds (reads
102 // the way to the end of the directory), it returns the slic
103 // nil error. If it encounters an error before the end of th
104 // directory, Readdir returns the FileInfo read until that p
105 // and a non-nil error.
106 func (f *File) Readdir(n int) (fi []FileInfo, err error) {
107     return f.readdir(n)
108 }
109
110 // Readdirnames reads and returns a slice of names from the
111 //
112 // If n > 0, Readdirnames returns at most n names. In this c
113 // Readdirnames returns an empty slice, it will return a non
114 // explaining why. At the end of a directory, the error is i
115 //
116 // If n <= 0, Readdirnames returns all the names from the di
117 // a single slice. In this case, if Readdirnames succeeds (r
118 // the way to the end of the directory), it returns the slic
119 // nil error. If it encounters an error before the end of th
120 // directory, Readdirnames returns the names read until that
121 // a non-nil error.
122 func (f *File) Readdirnames(n int) (names []string, err erro
123     return f.readdirnames(n)
124 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/env.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // General environment variables.
6
7 package os
8
9 import "syscall"
10
11 // Expand replaces ${var} or $var in the string based on the
12 // Invocations of undefined variables are replaced with the
13 func Expand(s string, mapping func(string) string) string {
14     buf := make([]byte, 0, 2*len(s))
15     // ${} is all ASCII, so bytes are fine for this oper
16     i := 0
17     for j := 0; j < len(s); j++ {
18         if s[j] == '$' && j+1 < len(s) {
19             buf = append(buf, s[i:j]...)
20             name, w := getShellName(s[j+1:])
21             buf = append(buf, mapping(name)...)
22             j += w
23             i = j + 1
24         }
25     }
26     return string(buf) + s[i:]
27 }
28
29 // ExpandEnv replaces ${var} or $var in the string according
30 // of the current environment variables. References to unde
31 // variables are replaced by the empty string.
32 func ExpandEnv(s string) string {
33     return Expand(s, Getenv)
34 }
35
36 // isSpellSpecialVar reports whether the character identifie
37 // shell variable such as $*.
38 func isShellSpecialVar(c uint8) bool {
39     switch c {
40     case '*', '#', '$', '@', '!', '?', '0', '1', '2', '3
41         return true
42     }
43     return false
44 }
```

```

45
46 // isAlphaNum reports whether the byte is an ASCII letter, n
47 func isAlphaNum(c uint8) bool {
48     return c == '_' || '0' <= c && c <= '9' || 'a' <= c
49 }
50
51 // getName returns the name that begins the string and the n
52 // consumed to extract it. If the name is enclosed in {}, i
53 // expansion and two more bytes are needed than the length o
54 func getShellName(s string) (string, int) {
55     switch {
56     case s[0] == '{':
57         if len(s) > 2 && isShellSpecialVar(s[1]) &&
58             return s[1:2], 3
59     }
60     // Scan to closing brace
61     for i := 1; i < len(s); i++ {
62         if s[i] == '}' {
63             return s[1:i], i + 1
64         }
65     }
66     return "", 1 // Bad syntax; just eat the bra
67     case isShellSpecialVar(s[0]):
68         return s[0:1], 1
69     }
70     // Scan alphanumerics.
71     var i int
72     for i = 0; i < len(s) && isAlphaNum(s[i]); i++ {
73     }
74     return s[:i], i
75 }
76
77 // Getenv retrieves the value of the environment variable na
78 // It returns the value, which will be empty if the variable
79 func Getenv(key string) string {
80     v, _ := syscall.Getenv(key)
81     return v
82 }
83
84 // Setenv sets the value of the environment variable named b
85 // It returns an error, if any.
86 func Setenv(key, value string) error {
87     err := syscall.Setenv(key, value)
88     if err != nil {
89         return NewSyscallError("setenv", err)
90     }
91     return nil
92 }
93
94 // Clearenv deletes all environment variables.

```

```
95 func Clearenv() {
96     syscall.Clearenv()
97 }
98
99 // Environ returns a copy of strings representing the environme
100 // in the form "key=value".
101 func Environ() []string {
102     return syscall.Environ()
103 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/error.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package os
6
7 import (
8     "errors"
9 )
10
11 // Portable analogs of some common system call errors.
12 var (
13     ErrInvalid      = errors.New("invalid argument")
14     ErrPermission   = errors.New("permission denied")
15     ErrExist        = errors.New("file already exists")
16     ErrNotExist     = errors.New("file does not exist")
17 )
18
19 // PathError records an error and the operation and file path.
20 type PathError struct {
21     Op    string
22     Path string
23     Err  error
24 }
25
26 func (e *PathError) Error() string { return e.Op + " " + e.Path + ": " + e.Err.Error() }
27
28 // SyscallError records an error from a specific system call.
29 type SyscallError struct {
30     Syscall string
31     Err     error
32 }
33
34 func (e *SyscallError) Error() string { return e.Syscall + " " + e.Err.Error() }
35
36 // NewSyscallError returns, as an error, a new SyscallError
37 // with the given system call name and error details.
38 // As a convenience, if err is nil, NewSyscallError returns
39 func NewSyscallError(syscall string, err error) error {
40     if err == nil {
41         return nil
42     }
43     return &SyscallError{syscall, err}
44 }
```

```
45
46 // IsExist returns whether the error is known to report that
47 // already exists. It is satisfied by ErrExist as well as so
48 func IsExist(err error) bool {
49     return isExist(err)
50 }
51
52 // IsNotExist returns whether the error is known to report t
53 // does not exist. It is satisfied by ErrNotExist as well as
54 func IsNotExist(err error) bool {
55     return isNotExist(err)
56 }
57
58 // IsPermission returns whether the error is known to report
59 // It is satisfied by ErrPermission as well as some syscall
60 func IsPermission(err error) bool {
61     return isPermission(err)
62 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/error_posix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package os
8
9 import "syscall"
10
11 func isExist(err error) bool {
12     if pe, ok := err.(*PathError); ok {
13         err = pe.Err
14     }
15     return err == syscall.EEXIST || err == ErrExist
16 }
17
18 func isNotExist(err error) bool {
19     if pe, ok := err.(*PathError); ok {
20         err = pe.Err
21     }
22     return err == syscall.ENOENT || err == ErrNotExist
23 }
24
25 func isPermission(err error) bool {
26     if pe, ok := err.(*PathError); ok {
27         err = pe.Err
28     }
29     return err == syscall.EACCES || err == syscall.EPERM
30 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/exec.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package os
6
7 import (
8     "runtime"
9     "syscall"
10 )
11
12 // Process stores the information about a process created by
13 type Process struct {
14     Pid      int
15     handle   uintptr
16     done     bool // process has been successfully waited
17 }
18
19 func newProcess(pid int, handle uintptr) *Process {
20     p := &Process{Pid: pid, handle: handle}
21     runtime.SetFinalizer(p, (*Process).Release)
22     return p
23 }
24
25 // ProcAttr holds the attributes that will be applied to a n
26 // started by StartProcess.
27 type ProcAttr struct {
28     // If Dir is non-empty, the child changes into the d
29     // creating the process.
30     Dir string
31     // If Env is non-nil, it gives the environment varia
32     // new process in the form returned by Environ.
33     // If it is nil, the result of Environ will be used.
34     Env []string
35     // Files specifies the open files inherited by the n
36     // first three entries correspond to standard input,
37     // standard error. An implementation may support ad
38     // depending on the underlying operating system. A
39     // to that file being closed when the process starts
40     Files []*File
41
42     // Operating system-specific process creation attrib
43     // Note that setting this field means that your prog
44     // may not execute properly or even compile on some
```

```

45         // operating systems.
46         Sys *syscall.SysProcAttr
47     }
48
49     // A Signal represents an operating system signal.
50     // The usual underlying implementation is operating system-d
51     // on Unix it is syscall.Signal.
52     type Signal interface {
53         String() string
54         Signal() // to distinguish from other Stringers
55     }
56
57     // The only signal values guaranteed to be present on all sy
58     // are Interrupt (send the process an interrupt) and
59     // Kill (force the process to exit).
60     var (
61         Interrupt Signal = syscall.SIGINT
62         Kill       Signal = syscall.SIGKILL
63     )
64
65     // Getpid returns the process id of the caller.
66     func Getpid() int { return syscall.Getpid() }
67
68     // Getppid returns the process id of the caller's parent.
69     func Getppid() int { return syscall.Getppid() }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/exec_posix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 package os
8
9 import (
10     "syscall"
11 )
12
13 func startProcess(name string, argv []string, attr *ProcAttr
14     // Double-check existence of the directory we want
15     // to chdir into. We can make the error clearer thi
16     if attr != nil && attr.Dir != "" {
17         if _, err := Stat(attr.Dir); err != nil {
18             pe := err.(*PathError)
19             pe.Op = "chdir"
20             return nil, pe
21         }
22     }
23
24     sysattr := &syscall.ProcAttr{
25         Dir: attr.Dir,
26         Env: attr.Env,
27         Sys: attr.Sys,
28     }
29     if sysattr.Env == nil {
30         sysattr.Env = Environ()
31     }
32     for _, f := range attr.Files {
33         sysattr.Files = append(sysattr.Files, f.Fd())
34     }
35
36     pid, h, e := syscall.StartProcess(name, argv, sysatt
37     if e != nil {
38         return nil, &PathError{"fork/exec", name, e}
39     }
40     return newProcess(pid, h), nil
41 }
42
43 func (p *Process) kill() error {
44     return p.Signal(Kill)
```

```

45 }
46
47 // ProcessState stores information about a process, as repor
48 type ProcessState struct {
49     pid    int           // The process's id.
50     status syscall.WaitStatus // System-dependent status
51     rusage *syscall.Rusage
52 }
53
54 // Pid returns the process id of the exited process.
55 func (p *ProcessState) Pid() int {
56     return p.pid
57 }
58
59 func (p *ProcessState) exited() bool {
60     return p.status.Exited()
61 }
62
63 func (p *ProcessState) success() bool {
64     return p.status.ExitStatus() == 0
65 }
66
67 func (p *ProcessState) sys() interface{} {
68     return p.status
69 }
70
71 func (p *ProcessState) sysUsage() interface{} {
72     return p.rusage
73 }
74
75 // Convert i to decimal string.
76 func itod(i int) string {
77     if i == 0 {
78         return "0"
79     }
80
81     u := uint64(i)
82     if i < 0 {
83         u = -u
84     }
85
86     // Assemble decimal in reverse order.
87     var b [32]byte
88     bp := len(b)
89     for ; u > 0; u /= 10 {
90         bp--
91         b[bp] = byte(u%10) + '0'
92     }
93
94     if i < 0 {

```

```

95         bp--
96         b[bp] = '-'
97     }
98
99     return string(b[bp:])
100 }
101
102 func (p *ProcessState) String() string {
103     if p == nil {
104         return "<nil>"
105     }
106     status := p.Sys().(syscall.WaitStatus)
107     res := ""
108     switch {
109     case status.Exited():
110         res = "exit status " + itod(status.ExitStatus)
111     case status.Signaled():
112         res = "signal " + itod(int(status.Signal()))
113     case status.Stopped():
114         res = "stop signal " + itod(int(status.StopSignal))
115         if status.StopSignal() == syscall.SIGTRAP &&
116             res += " (trap " + itod(status.TrapCause)
117     }
118     case status.Continued():
119         res = "continued"
120     }
121     if status.CoreDump() {
122         res += " (core dumped)"
123     }
124     return res
125 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/exec_unix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package os
8
9 import (
10     "errors"
11     "runtime"
12     "syscall"
13     "time"
14 )
15
16 func (p *Process) wait() (ps *ProcessState, err error) {
17     if p.Pid == -1 {
18         return nil, syscall.EINVAL
19     }
20     var status syscall.WaitStatus
21     var rusage syscall.Rusage
22     pid1, e := syscall.Wait4(p.Pid, &status, 0, &rusage)
23     if e != nil {
24         return nil, NewSyscallError("wait", e)
25     }
26     if pid1 != 0 {
27         p.done = true
28     }
29     ps = &ProcessState{
30         pid:    pid1,
31         status: status,
32         rusage: &rusage,
33     }
34     return ps, nil
35 }
36
37 func (p *Process) signal(sig Signal) error {
38     if p.done {
39         return errors.New("os: process already finish")
40     }
41     s, ok := sig.(syscall.Signal)
42     if !ok {
43         return errors.New("os: unsupported signal type")
44     }
45 }
```

```

45         if e := syscall.Kill(p.Pid, s); e != nil {
46             return e
47         }
48         return nil
49     }
50
51     func (p *Process) release() error {
52         // NOOP for unix.
53         p.Pid = -1
54         // no need for a finalizer anymore
55         runtime.SetFinalizer(p, nil)
56         return nil
57     }
58
59     func findProcess(pid int) (p *Process, err error) {
60         // NOOP for unix.
61         return newProcess(pid, 0), nil
62     }
63
64     func (p *ProcessState) userTime() time.Duration {
65         return time.Duration(p.rusage.Utime.Nano()) * time.N
66     }
67
68     func (p *ProcessState) systemTime() time.Duration {
69         return time.Duration(p.rusage.Stime.Nano()) * time.N
70     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/file.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package os provides a platform-independent interface to o
6 // functionality. The design is Unix-like, although the erro
7 // Go-like; failing calls return values of type error rather
8 // Often, more information is available within the error. Fo
9 // if a call that takes a file name fails, such as Open or S
10 // will include the failing file name when printed and will
11 // *PathError, which may be unpacked for more information.
12 //
13 // The os interface is intended to be uniform across all ope
14 // Features not generally available appear in the system-spe
15 //
16 // Here is a simple example, opening a file and reading some
17 //
18 //     file, err := os.Open("file.go") // For read access.
19 //     if err != nil {
20 //         log.Fatal(err)
21 //     }
22 //
23 // If the open fails, the error string will be self-explanat
24 //
25 //     open file.go: no such file or directory
26 //
27 // The file's data can then be read into a slice of bytes. R
28 // Write take their byte counts from the length of the argum
29 //
30 //     data := make([]byte, 100)
31 //     count, err := file.Read(data)
32 //     if err != nil {
33 //         log.Fatal(err)
34 //     }
35 //     fmt.Printf("read %d bytes: %q\n", count, data[:count
36 //
37 package os
38
39 import (
40     "io"
41     "syscall"
42 )
43
44 // Name returns the name of the file as presented to Open.
```

```

45 func (f *File) Name() string { return f.name }
46
47 // Stdin, Stdout, and Stderr are open Files pointing to the
48 // standard output, and standard error file descriptors.
49 var (
50     Stdin = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
51     Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
52     Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
53 )
54
55 // Flags to Open wrapping those of the underlying system. No
56 // may be implemented on a given system.
57 const (
58     O_RDONLY int = syscall.O_RDONLY // open the file read-only
59     O_WRONLY int = syscall.O_WRONLY // open the file write-only
60     O_RDWR  int = syscall.O_RDWR   // open the file read-write
61     O_APPEND int = syscall.O_APPEND // append data to the file
62     O_CREATE int = syscall.O_CREAT  // create a new file if none exist
63     O_EXCL   int = syscall.O_EXCL   // used with O_CREAT to fail if
64     O_SYNC   int = syscall.O_SYNC   // open for synchronous I/O
65     O_TRUNC  int = syscall.O_TRUNC  // if possible, truncate the file
66 )
67
68 // Seek whence values.
69 const (
70     SEEK_SET int = 0 // seek relative to the origin of the file
71     SEEK_CUR int = 1 // seek relative to the current offset
72     SEEK_END int = 2 // seek relative to the end of the file
73 )
74
75 // LinkError records an error during a link or symlink or rename
76 // system call and the paths that caused it.
77 type LinkError struct {
78     Op string
79     Old string
80     New string
81     Err error
82 }
83
84 func (e *LinkError) Error() string {
85     return e.Op + " " + e.Old + " " + e.New + ": " + e.Err.Error()
86 }
87
88 // Read reads up to len(b) bytes from the File.
89 // It returns the number of bytes read and an error, if any.
90 // EOF is signaled by a zero count with err set to io.EOF.
91 func (f *File) Read(b []byte) (n int, err error) {
92     if f == nil {
93         return 0, ErrInvalid
94     }

```

```

95         n, e := f.read(b)
96         if n < 0 {
97             n = 0
98         }
99         if n == 0 && len(b) > 0 && e == nil {
100             return 0, io.EOF
101         }
102         if e != nil {
103             err = &PathError{"read", f.name, e}
104         }
105         return n, err
106     }
107
108     // ReadAt reads len(b) bytes from the File starting at byte
109     // It returns the number of bytes read and the error, if any
110     // ReadAt always returns a non-nil error when n < len(b).
111     // At end of file, that error is io.EOF.
112     func (f *File) ReadAt(b []byte, off int64) (n int, err error) {
113         if f == nil {
114             return 0, ErrInvalid
115         }
116         for len(b) > 0 {
117             m, e := f.pread(b, off)
118             if m == 0 && e == nil {
119                 return n, io.EOF
120             }
121             if e != nil {
122                 err = &PathError{"read", f.name, e}
123                 break
124             }
125             n += m
126             b = b[m:]
127             off += int64(m)
128         }
129         return
130     }
131
132     // Write writes len(b) bytes to the File.
133     // It returns the number of bytes written and an error, if a
134     // Write returns a non-nil error when n != len(b).
135     func (f *File) Write(b []byte) (n int, err error) {
136         if f == nil {
137             return 0, ErrInvalid
138         }
139         n, e := f.write(b)
140         if n < 0 {
141             n = 0
142         }
143

```

```

144         epipecheck(f, e)
145
146         if e != nil {
147             err = &PathError{"write", f.name, e}
148         }
149         return n, err
150     }
151
152     // WriteAt writes len(b) bytes to the File starting at byte
153     // It returns the number of bytes written and an error, if a
154     // WriteAt returns a non-nil error when n != len(b).
155     func (f *File) WriteAt(b []byte, off int64) (n int, err error) {
156         if f == nil {
157             return 0, ErrInvalid
158         }
159         for len(b) > 0 {
160             m, e := f.pwrite(b, off)
161             if e != nil {
162                 err = &PathError{"write", f.name, e}
163                 break
164             }
165             n += m
166             b = b[m:]
167             off += int64(m)
168         }
169         return
170     }
171
172     // Seek sets the offset for the next Read or Write on file t
173     // according to whence: 0 means relative to the origin of the
174     // file, 1 means relative to the current offset, and 2 means relative to the
175     // end of the file. It returns the new offset and an error, if any.
176     func (f *File) Seek(offset int64, whence int) (ret int64, err error) {
177         r, e := f.seek(offset, whence)
178         if e == nil && f.dirinfo != nil && r != 0 {
179             e = syscall.EISDIR
180         }
181         if e != nil {
182             return 0, &PathError{"seek", f.name, e}
183         }
184         return r, nil
185     }
186
187     // WriteString is like Write, but writes the contents of str
188     // an array of bytes.
189     func (f *File) WriteString(s string) (ret int, err error) {
190         if f == nil {
191             return 0, ErrInvalid
192         }

```

```

193         return f.Write([]byte(s))
194     }
195
196     // Mkdir creates a new directory with the specified name and
197     // If there is an error, it will be of type *PathError.
198     func Mkdir(name string, perm FileMode) error {
199         e := syscall.Mkdir(name, syscallMode(perm))
200         if e != nil {
201             return &PathError{"mkdir", name, e}
202         }
203         return nil
204     }
205
206     // Chdir changes the current working directory to the named
207     // If there is an error, it will be of type *PathError.
208     func Chdir(dir string) error {
209         if e := syscall.Chdir(dir); e != nil {
210             return &PathError{"chdir", dir, e}
211         }
212         return nil
213     }
214
215     // Chdir changes the current working directory to the file,
216     // which must be a directory.
217     // If there is an error, it will be of type *PathError.
218     func (f *File) Chdir() error {
219         if e := syscall.Fchdir(f.fd); e != nil {
220             return &PathError{"chdir", f.name, e}
221         }
222         return nil
223     }
224
225     // Open opens the named file for reading. If successful, me
226     // the returned file can be used for reading; the associated
227     // descriptor has mode O_RDONLY.
228     // If there is an error, it will be of type *PathError.
229     func Open(name string) (file *File, err error) {
230         return OpenFile(name, O_RDONLY, 0)
231     }
232
233     // Create creates the named file mode 0666 (before umask), t
234     // it if it already exists. If successful, methods on the r
235     // File can be used for I/O; the associated file descriptor
236     // O_RDWR.
237     // If there is an error, it will be of type *PathError.
238     func Create(name string) (file *File, err error) {
239         return OpenFile(name, O_RDWR|O_CREATE|O_TRUNC, 0666)
240     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/file_posix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 package os
8
9 import (
10     "syscall"
11     "time"
12 )
13
14 func sigpipe() // implemented in package runtime
15
16 func epipecheck(file *File, e error) {
17     if e == syscall.EPIPE {
18         file.nepipe++
19         if file.nepipe >= 10 {
20             sigpipe()
21         }
22     } else {
23         file.nepipe = 0
24     }
25 }
26
27 // Link creates newname as a hard link to the oldname file.
28 // If there is an error, it will be of type *LinkError.
29 func Link(oldname, newname string) error {
30     e := syscall.Link(oldname, newname)
31     if e != nil {
32         return &LinkError{"link", oldname, newname,
33     }
34     return nil
35 }
36
37 // Symlink creates newname as a symbolic link to oldname.
38 // If there is an error, it will be of type *LinkError.
39 func Symlink(oldname, newname string) error {
40     e := syscall.Symlink(oldname, newname)
41     if e != nil {
42         return &LinkError{"symlink", oldname, newname,
43     }
44     return nil

```

```

45 }
46
47 // Readlink returns the destination of the named symbolic li
48 // If there is an error, it will be of type *PathError.
49 func Readlink(name string) (string, error) {
50     for len := 128; ; len *= 2 {
51         b := make([]byte, len)
52         n, e := syscall.Readlink(name, b)
53         if e != nil {
54             return "", &PathError{"readlink", na
55         }
56         if n < len {
57             return string(b[0:n]), nil
58         }
59     }
60     // Silence 6g.
61     return "", nil
62 }
63
64 // Rename renames a file.
65 func Rename(oldname, newname string) error {
66     e := syscall.Rename(oldname, newname)
67     if e != nil {
68         return &LinkError{"rename", oldname, newname
69     }
70     return nil
71 }
72
73 // syscallMode returns the syscall-specific mode bits from G
74 func syscallMode(i FileMode) (o uint32) {
75     o |= uint32(i.Perm())
76     if i&ModeSetuid != 0 {
77         o |= syscall.S_ISUID
78     }
79     if i&ModeSetgid != 0 {
80         o |= syscall.S_ISGID
81     }
82     if i&ModeSticky != 0 {
83         o |= syscall.S_ISVTX
84     }
85     // No mapping for Go's ModeTemporary (plan9 only).
86     return
87 }
88
89 // Chmod changes the mode of the named file to mode.
90 // If the file is a symbolic link, it changes the mode of th
91 // If there is an error, it will be of type *PathError.
92 func Chmod(name string, mode FileMode) error {
93     if e := syscall.Chmod(name, syscallMode(mode)); e !=
94         return &PathError{"chmod", name, e}

```

```

95         }
96         return nil
97     }
98
99     // Chmod changes the mode of the file to mode.
100    // If there is an error, it will be of type *PathError.
101    func (f *File) Chmod(mode FileMode) error {
102        if e := syscall.Fchmod(f.fd, syscallMode(mode)); e != nil {
103            return &PathError{"chmod", f.name, e}
104        }
105        return nil
106    }
107
108    // Chown changes the numeric uid and gid of the named file.
109    // If the file is a symbolic link, it changes the uid and gi
110    // If there is an error, it will be of type *PathError.
111    func Chown(name string, uid, gid int) error {
112        if e := syscall.Chown(name, uid, gid); e != nil {
113            return &PathError{"chown", name, e}
114        }
115        return nil
116    }
117
118    // Lchown changes the numeric uid and gid of the named file.
119    // If the file is a symbolic link, it changes the uid and gi
120    // If there is an error, it will be of type *PathError.
121    func Lchown(name string, uid, gid int) error {
122        if e := syscall.Lchown(name, uid, gid); e != nil {
123            return &PathError{"lchown", name, e}
124        }
125        return nil
126    }
127
128    // Chown changes the numeric uid and gid of the named file.
129    // If there is an error, it will be of type *PathError.
130    func (f *File) Chown(uid, gid int) error {
131        if e := syscall.Fchown(f.fd, uid, gid); e != nil {
132            return &PathError{"chown", f.name, e}
133        }
134        return nil
135    }
136
137    // Truncate changes the size of the file.
138    // It does not change the I/O offset.
139    // If there is an error, it will be of type *PathError.
140    func (f *File) Truncate(size int64) error {
141        if e := syscall.Ftruncate(f.fd, size); e != nil {
142            return &PathError{"truncate", f.name, e}
143        }

```

```

144         return nil
145     }
146
147     // Sync commits the current contents of the file to stable s
148     // Typically, this means flushing the file system's in-memor
149     // of recently written data to disk.
150     func (f *File) Sync() (err error) {
151         if f == nil {
152             return syscall.EINVAL
153         }
154         if e := syscall.Fsync(f.fd); e != nil {
155             return NewSyscallError("fsync", e)
156         }
157         return nil
158     }
159
160     // Chtimes changes the access and modification times of the
161     // file, similar to the Unix utime() or utimes() functions.
162     //
163     // The underlying filesystem may truncate or round the value
164     // less precise time unit.
165     // If there is an error, it will be of type *PathError.
166     func Chtimes(name string, atime time.Time, mtime time.Time)
167         var utimes [2]syscall.Timeval
168         atime_ns := atime.Unix()*1e9 + int64(atime.Nanosecon
169         mtime_ns := mtime.Unix()*1e9 + int64(mtime.Nanosecon
170         utimes[0] = syscall.NsecToTimeval(atime_ns)
171         utimes[1] = syscall.NsecToTimeval(mtime_ns)
172         if e := syscall.Utimes(name, utimes[0:]); e != nil {
173             return &PathError{"chtimes", name, e}
174         }
175         return nil
176     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/file_unix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package os
8
9 import (
10     "runtime"
11     "syscall"
12 )
13
14 // File represents an open file descriptor.
15 type File struct {
16     *file
17 }
18
19 // file is the real representation of *File.
20 // The extra level of indirection ensures that no clients of
21 // can overwrite this data, which could cause the finalizer
22 // to close the wrong file descriptor.
23 type file struct {
24     fd      int
25     name    string
26     dirinfo *dirInfo // nil unless directory being read
27     nepipe  int      // number of consecutive EPIPE in w
28 }
29
30 // Fd returns the integer Unix file descriptor referencing t
31 func (f *File) Fd() uintptr {
32     if f == nil {
33         return ^(uintptr(0))
34     }
35     return uintptr(f.fd)
36 }
37
38 // NewFile returns a new File with the given file descriptor
39 func NewFile(fd uintptr, name string) *File {
40     fdi := int(fd)
41     if fdi < 0 {
42         return nil
43     }
44     f := &File{&file{fd: fdi, name: name}}
```

```

45         runtime.SetFinalizer(f.file, (*file).close)
46         return f
47     }
48
49     // Auxiliary information if the File describes a directory
50     type dirInfo struct {
51         buf []byte // buffer for directory I/O
52         nbuf int    // length of buf; return value from Getd
53         bufp int    // location of next record in buf.
54     }
55
56     // DevNull is the name of the operating system's ``null devi
57     // On Unix-like systems, it is "/dev/null"; on Windows, "NUL
58     const DevNull = "/dev/null"
59
60     // OpenFile is the generalized open call; most users will us
61     // or Create instead. It opens the named file with specifie
62     // (O_RDONLY etc.) and perm, (0666 etc.) if applicable. If
63     // methods on the returned File can be used for I/O.
64     // If there is an error, it will be of type *PathError.
65     func OpenFile(name string, flag int, perm FileMode) (file *F
66         r, e := syscall.Open(name, flag|syscall.O_CLOEXEC, s
67         if e != nil {
68             return nil, &PathError{"open", name, e}
69         }
70
71         // There's a race here with fork/exec, which we are
72         // content to live with. See ../syscall/exec_unix.g
73         // On OS X 10.6, the O_CLOEXEC flag is not respected
74         // On OS X 10.7, the O_CLOEXEC flag works.
75         // Without a cheap & reliable way to detect 10.6 vs
76         // runtime, we just always call syscall.CloseOnExec
77         // Once >=10.7 is prevalent, this extra call can rem
78         if syscall.O_CLOEXEC == 0 || runtime.GOOS == "darwin
79             syscall.CloseOnExec(r)
80         }
81
82         return NewFile(uintptr(r), name), nil
83     }
84
85     // Close closes the File, rendering it unusable for I/O.
86     // It returns an error, if any.
87     func (f *File) Close() error {
88         return f.file.close()
89     }
90
91     func (file *file) close() error {
92         if file == nil || file.fd < 0 {
93             return syscall.EINVAL
94         }

```

```

95     var err error
96     if e := syscall.Close(file.fd); e != nil {
97         err = &PathError{"close", file.name, e}
98     }
99     file.fd = -1 // so it can't be closed again
100
101     // no need for a finalizer anymore
102     runtime.SetFinalizer(file, nil)
103     return err
104 }
105
106 // Stat returns the FileInfo structure describing file.
107 // If there is an error, it will be of type *PathError.
108 func (f *File) Stat() (fi FileInfo, err error) {
109     var stat syscall.Stat_t
110     err = syscall.Fstat(f.fd, &stat)
111     if err != nil {
112         return nil, &PathError{"stat", f.name, err}
113     }
114     return fileInfoFromStat(&stat, f.name), nil
115 }
116
117 // Stat returns a FileInfo describing the named file.
118 // If there is an error, it will be of type *PathError.
119 func Stat(name string) (fi FileInfo, err error) {
120     var stat syscall.Stat_t
121     err = syscall.Stat(name, &stat)
122     if err != nil {
123         return nil, &PathError{"stat", name, err}
124     }
125     return fileInfoFromStat(&stat, name), nil
126 }
127
128 // Lstat returns a FileInfo describing the named file.
129 // If the file is a symbolic link, the returned FileInfo
130 // describes the symbolic link. Lstat makes no attempt to f
131 // If there is an error, it will be of type *PathError.
132 func Lstat(name string) (fi FileInfo, err error) {
133     var stat syscall.Stat_t
134     err = syscall.Lstat(name, &stat)
135     if err != nil {
136         return nil, &PathError{"lstat", name, err}
137     }
138     return fileInfoFromStat(&stat, name), nil
139 }
140
141 func (f *File) readdir(n int) (fi []FileInfo, err error) {
142     dirname := f.name
143     if dirname == "" {

```

```

144         dirname = "."
145     }
146     dirname += "/"
147     names, err := f.Readdirnames(n)
148     fi = make([]FileInfo, len(names))
149     for i, filename := range names {
150         fip, err := Lstat(dirname + filename)
151         if err == nil {
152             fi[i] = fip
153         } else {
154             fi[i] = &fileStat{name: filename}
155         }
156     }
157     return fi, err
158 }
159
160 // read reads up to len(b) bytes from the File.
161 // It returns the number of bytes read and an error, if any.
162 func (f *File) read(b []byte) (n int, err error) {
163     return syscall.Read(f.fd, b)
164 }
165
166 // pread reads len(b) bytes from the File starting at byte o
167 // It returns the number of bytes read and the error, if any
168 // EOF is signaled by a zero count with err set to 0.
169 func (f *File) pread(b []byte, off int64) (n int, err error) {
170     return syscall.Pread(f.fd, b, off)
171 }
172
173 // write writes len(b) bytes to the File.
174 // It returns the number of bytes written and an error, if a
175 func (f *File) write(b []byte) (n int, err error) {
176     for {
177         m, err := syscall.Write(f.fd, b)
178         n += m
179
180         // If the syscall wrote some data but not all
181         // or it returned EINTR, then assume it stop
182         // reasons that are uninteresting to the call
183         if 0 < m && m < len(b) || err == syscall.EIN
184             b = b[m:]
185             continue
186     }
187
188     return n, err
189 }
190 panic("not reached")
191 }
192

```

```

193 // pwrite writes len(b) bytes to the File starting at byte o
194 // It returns the number of bytes written and an error, if a
195 func (f *File) pwrite(b []byte, off int64) (n int, err error)
196     return syscall.Pwrite(f.fd, b, off)
197 }
198
199 // seek sets the offset for the next Read or Write on file t
200 // according to whence: 0 means relative to the origin of th
201 // relative to the current offset, and 2 means relative to t
202 // It returns the new offset and an error, if any.
203 func (f *File) seek(offset int64, whence int) (ret int64, er
204     return syscall.Seek(f.fd, offset, whence)
205 }
206
207 // Truncate changes the size of the named file.
208 // If the file is a symbolic link, it changes the size of th
209 // If there is an error, it will be of type *PathError.
210 func Truncate(name string, size int64) error {
211     if e := syscall.Truncate(name, size); e != nil {
212         return &PathError{"truncate", name, e}
213     }
214     return nil
215 }
216
217 // Remove removes the named file or directory.
218 // If there is an error, it will be of type *PathError.
219 func Remove(name string) error {
220     // System call interface forces us to know
221     // whether name is a file or directory.
222     // Try both: it is cheaper on average than
223     // doing a Stat plus the right one.
224     e := syscall.Unlink(name)
225     if e == nil {
226         return nil
227     }
228     e1 := syscall.Rmdir(name)
229     if e1 == nil {
230         return nil
231     }
232
233     // Both failed: figure out which error to return.
234     // OS X and Linux differ on whether unlink(dir)
235     // returns EISDIR, so can't use that. However,
236     // both agree that rmdir(file) returns ENOTDIR,
237     // so we can use that to decide which error is real.
238     // Rmdir might also return ENOTDIR if given a bad
239     // file path, like /etc/passwd/foo, but in that case
240     // both errors will be ENOTDIR, so it's okay to
241     // use the error from unlink.
242     if e1 != syscall.ENOTDIR {

```

```

243             e = e1
244         }
245         return &PathError{"remove", name, e}
246     }
247
248     // basename removes trailing slashes and the leading directory
249     func basename(name string) string {
250         i := len(name) - 1
251         // Remove trailing slashes
252         for ; i > 0 && name[i] == '/'; i-- {
253             name = name[:i]
254         }
255         // Remove leading directory name
256         for i--; i >= 0; i-- {
257             if name[i] == '/' {
258                 name = name[i+1:]
259                 break
260             }
261         }
262
263         return name
264     }
265
266     // Pipe returns a connected pair of Files; reads from r return
267     // It returns the files and an error, if any.
268     func Pipe() (r *File, w *File, err error) {
269         var p [2]int
270
271         // See ../syscall/exec.go for description of lock.
272         syscall.ForkLock.RLock()
273         e := syscall.Pipe(p[0:])
274         if e != nil {
275             syscall.ForkLock.RUnlock()
276             return nil, nil, NewSyscallError("pipe", e)
277         }
278         syscall.CloseOnExec(p[0])
279         syscall.CloseOnExec(p[1])
280         syscall.ForkLock.RUnlock()
281
282         return NewFile(uintptr(p[0]), "|0"), NewFile(uintptr
283     }
284
285     // TempDir returns the default directory to use for temporary
286     func TempDir() string {
287         dir := Getenv("TMPDIR")
288         if dir == "" {
289             dir = "/tmp"
290         }
291         return dir

```

292 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/getwd.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package os
6
7 import (
8     "syscall"
9 )
10
11 // Getwd returns a rooted path name corresponding to the
12 // current directory. If the current directory can be
13 // reached via multiple paths (due to symbolic links),
14 // Getwd may return any one of them.
15 func Getwd() (pwd string, err error) {
16     // If the operating system provides a Getwd call, use it.
17     if syscall.ImplementsGetwd {
18         s, e := syscall.Getwd()
19         return s, NewSyscallError("getwd", e)
20     }
21
22     // Otherwise, we're trying to find our way back to "."
23     dot, err := Stat(".")
24     if err != nil {
25         return "", err
26     }
27
28     // Clumsy but widespread kludge:
29     // if $PWD is set and matches ".", use it.
30     pwd = Getenv("PWD")
31     if len(pwd) > 0 && pwd[0] == '/' {
32         d, err := Stat(pwd)
33         if err == nil && SameFile(dot, d) {
34             return pwd, nil
35         }
36     }
37
38     // Root is a special case because it has no parent
39     // and ends in a slash.
40     root, err := Stat("/")
41     if err != nil {
42         // Can't stat root - no hope.
43         return "", err
44     }
45 }
```

```

45     if SameFile(root, dot) {
46         return "/", nil
47     }
48
49     // General algorithm: find name in parent
50     // and then find name of parent. Each iteration
51     // adds /name to the beginning of pwd.
52     pwd = ""
53     for parent := ".."; ; parent = "../" + parent {
54         if len(parent) >= 1024 { // Sanity check
55             return "", syscall.ENAMETOOLONG
56         }
57         fd, err := Open(parent)
58         if err != nil {
59             return "", err
60         }
61
62         for {
63             names, err := fd.Readdirnames(100)
64             if err != nil {
65                 fd.Close()
66                 return "", err
67             }
68             for _, name := range names {
69                 d, _ := Lstat(parent + "/" +
70                     name)
71                 if SameFile(d, dot) {
72                     pwd = "/" + name + p
73                     goto Found
74                 }
75             }
76             fd.Close()
77             return "", ErrNotExist
78
79         Found:
80             pd, err := fd.Stat()
81             if err != nil {
82                 return "", err
83             }
84             fd.Close()
85             if SameFile(pd, root) {
86                 break
87             }
88             // Set up for next round.
89             dot = pd
90         }
91     return pwd, nil
92 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/path.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package os
6
7 import (
8     "io"
9     "syscall"
10 )
11
12 // MkdirAll creates a directory named path,
13 // along with any necessary parents, and returns nil,
14 // or else returns an error.
15 // The permission bits perm are used for all
16 // directories that MkdirAll creates.
17 // If path is already a directory, MkdirAll does nothing
18 // and returns nil.
19 func MkdirAll(path string, perm FileMode) error {
20     // If path exists, stop with success or error.
21     dir, err := Stat(path)
22     if err == nil {
23         if dir.IsDir() {
24             return nil
25         }
26         return &PathError{"mkdir", path, syscall.ENO}
27     }
28
29     // Doesn't already exist; make sure parent does.
30     i := len(path)
31     for i > 0 && IsPathSeparator(path[i-1]) { // Skip trailing
32         i--
33     }
34
35     j := i
36     for j > 0 && !IsPathSeparator(path[j-1]) { // Scan back to
37         j--
38     }
39
40     if j > 1 {
41         // Create parent
42         err = MkdirAll(path[0:j-1], perm)
43         if err != nil {
44             return err
45         }
46     }
47
48     // Create this directory.
49     err = Mkdir(path, perm)
50     if err != nil {
51         return err
52     }
53
54     return nil
55 }
```

```

45         }
46     }
47
48     // Now parent exists, try to create.
49     err = Mkdir(path, perm)
50     if err != nil {
51         // Handle arguments like "foo/." by
52         // double-checking that directory doesn't ex
53         dir, err1 := Lstat(path)
54         if err1 == nil && dir.IsDir() {
55             return nil
56         }
57         return err
58     }
59     return nil
60 }
61
62 // RemoveAll removes path and any children it contains.
63 // It removes everything it can but returns the first error
64 // it encounters. If the path does not exist, RemoveAll
65 // returns nil (no error).
66 func RemoveAll(path string) error {
67     // Simple case: if Remove works, we're done.
68     err := Remove(path)
69     if err == nil {
70         return nil
71     }
72
73     // Otherwise, is this a directory we need to recurse
74     dir, serr := Lstat(path)
75     if serr != nil {
76         if serr, ok := serr.(*PathError); ok && (IsN
77             return nil
78         }
79         return serr
80     }
81     if !dir.IsDir() {
82         // Not a directory; return the error from Re
83         return err
84     }
85
86     // Directory.
87     fd, err := Open(path)
88     if err != nil {
89         return err
90     }
91
92     // Remove contents & return first error.
93     err = nil
94     for {

```

```

95         names, err1 := fd.Readdirnames(100)
96         for _, name := range names {
97             err1 := RemoveAll(path + string(Path
98                 if err == nil {
99                     err = err1
100                }
101            }
102            if err1 == io.EOF {
103                break
104            }
105            // If Readdirnames returned an error, use it
106            if err == nil {
107                err = err1
108            }
109            if len(names) == 0 {
110                break
111            }
112        }
113
114        // Close directory, because windows won't remove ope
115        fd.Close()
116
117        // Remove directory.
118        err1 := Remove(path)
119        if err == nil {
120            err = err1
121        }
122        return err
123    }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/path_unix.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package os
8
9 const (
10     PathSeparator      = '/' // OS-specific path separator
11     PathListSeparator = ':' // OS-specific path list separator
12 )
13
14 // IsPathSeparator returns true if c is a directory separator
15 func IsPathSeparator(c uint8) bool {
16     return PathSeparator == c
17 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/proc.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Process etc.
6
7 package os
8
9 import "syscall"
10
11 // Args hold the command-line arguments, starting with the p
12 var Args []string
13
14 // Getuid returns the numeric user id of the caller.
15 func Getuid() int { return syscall.Getuid() }
16
17 // Geteuid returns the numeric effective user id of the call
18 func Geteuid() int { return syscall.Geteuid() }
19
20 // Getgid returns the numeric group id of the caller.
21 func Getgid() int { return syscall.Getgid() }
22
23 // Getegid returns the numeric effective group id of the cal
24 func Getegid() int { return syscall.Getegid() }
25
26 // Getgroups returns a list of the numeric ids of groups tha
27 func Getgroups() ([]int, error) {
28     gids, e := syscall.Getgroups()
29     return gids, NewSyscallError("getgroups", e)
30 }
31
32 // Exit causes the current program to exit with the given st
33 // Conventionally, code zero indicates success, non-zero an
34 func Exit(code int) { syscall.Exit(code) }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/stat_linux.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package os
6
7 import (
8     "syscall"
9     "time"
10 )
11
12 func sameFile(sys1, sys2 interface{}) bool {
13     stat1 := sys1.(*syscall.Stat_t)
14     stat2 := sys2.(*syscall.Stat_t)
15     return stat1.Dev == stat2.Dev && stat1.Ino == stat2.
16 }
17
18 func fileInfoFromStat(st *syscall.Stat_t, name string) FileI
19     fs := &fileStat{
20         name:    basename(name),
21         size:    int64(st.Size),
22         modTime: timespecToTime(st.Mtim),
23         sys:     st,
24     }
25     fs.mode = FileMode(st.Mode & 0777)
26     switch st.Mode & syscall.S_IFMT {
27     case syscall.S_IFBLK:
28         fs.mode |= ModeDevice
29     case syscall.S_IFCHR:
30         fs.mode |= ModeDevice | ModeCharDevice
31     case syscall.S_IFDIR:
32         fs.mode |= ModeDir
33     case syscall.S_IFIFO:
34         fs.mode |= ModeNamedPipe
35     case syscall.S_IFLNK:
36         fs.mode |= ModeSymlink
37     case syscall.S_IFREG:
38         // nothing to do
39     case syscall.S_IFSOCK:
40         fs.mode |= ModeSocket
41     }
42     if st.Mode&syscall.S_ISGID != 0 {
43         fs.mode |= ModeSetgid
44     }
```

```
45         if st.Mode&syscall.S_ISUID != 0 {
46             fs.mode |= ModeSetuid
47         }
48         if st.Mode&syscall.S_ISVTX != 0 {
49             fs.mode |= ModeSticky
50         }
51         return fs
52     }
53
54     func timespecToTime(ts syscall.Timespec) time.Time {
55         return time.Unix(int64(ts.Sec), int64(ts.Nsec))
56     }
57
58     // For testing.
59     func atime(fi FileInfo) time.Time {
60         return timespecToTime(fi.Sys().(*syscall.Stat_t).Ati
61     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/sys_linux.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Linux-specific
6
7 package os
8
9 func hostname() (name string, err error) {
10     f, err := Open("/proc/sys/kernel/hostname")
11     if err != nil {
12         return "", err
13     }
14     defer f.Close()
15
16     var buf [512]byte // Enough for a DNS name.
17     n, err := f.Read(buf[0:])
18     if err != nil {
19         return "", err
20     }
21
22     if n > 0 && buf[n-1] == '\n' {
23         n--
24     }
25     return string(buf[0:n]), nil
26 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/types.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package os
6
7 import (
8     "syscall"
9     "time"
10 )
11
12 // Getpagesize returns the underlying system's memory page size.
13 func Getpagesize() int { return syscall.Getpagesize() }
14
15 // A FileInfo describes a file and is returned by Stat and Lstat.
16 type FileInfo interface {
17     Name() string           // base name of the file
18     Size() int64           // length in bytes for regular files
19     Mode() FileMode       // file mode bits
20     ModTime() time.Time  // modification time
21     IsDir() bool         // abbreviation for Mode().IsDir()
22     Sys() interface{}    // underlying data source (can return nil)
23 }
24
25 // A FileMode represents a file's mode and permission bits.
26 // The bits have the same definition on all systems, so that
27 // information about files can be moved from one system
28 // to another portably. Not all bits apply to all systems.
29 // The only required bit is ModeDir for directories.
30 type FileMode uint32
31
32 // The defined file mode bits are the most significant bits
33 // The nine least-significant bits are the standard Unix rwx
34 // The values of these bits should be considered part of the
35 // may be used in wire protocols or disk representations: they
36 // changed, although new bits might be added.
37 const (
38     // The single letters are the abbreviations
39     // used by the String method's formatting.
40     ModeDir      FileMode = 1 << (32 - 1 - iota) // d:
41     ModeAppend   // a:
42     ModeExclusive // l:
43     ModeTemporary // T:
44     ModeSymlink  // L:
```

```

45     ModeDevice                // D:
46     ModeNamedPipe            // p:
47     ModeSocket                // S:
48     ModeSetuid                // u:
49     ModeSetgid                // g:
50     ModeCharDevice           // c:
51     ModeSticky                // t:
52
53     // Mask for the type bits. For regular files, none w
54     ModeType = ModeDir | ModeSymlink | ModeNamedPipe | M
55
56     ModePerm FileMode = 0777 // permission bits
57 )
58
59 func (m FileMode) String() string {
60     const str = "daLTLDpSugct"
61     var buf [32]byte // Mode is uint32.
62     w := 0
63     for i, c := range str {
64         if m&(1<<uint(32-1-i)) != 0 {
65             buf[w] = byte(c)
66             w++
67         }
68     }
69     if w == 0 {
70         buf[w] = '-'
71         w++
72     }
73     const rwx = "rwxrwxrwx"
74     for i, c := range rwx {
75         if m&(1<<uint(9-1-i)) != 0 {
76             buf[w] = byte(c)
77         } else {
78             buf[w] = '-'
79         }
80         w++
81     }
82     return string(buf[:w])
83 }
84
85 // IsDir reports whether m describes a directory.
86 // That is, it tests for the ModeDir bit being set in m.
87 func (m FileMode) IsDir() bool {
88     return m&ModeDir != 0
89 }
90
91 // Perm returns the Unix permission bits in m.
92 func (m FileMode) Perm() FileMode {
93     return m & ModePerm
94 }

```

```

95
96 // A fileStat is the implementation of FileInfo returned by
97 type fileStat struct {
98     name    string
99     size    int64
100    mode    FileMode
101    modTime time.Time
102    sys     interface{}
103 }
104
105 func (fs *fileStat) Name() string      { return fs.name }
106 func (fs *fileStat) Size() int64      { return fs.size }
107 func (fs *fileStat) Mode() FileMode   { return fs.mode }
108 func (fs *fileStat) ModTime() time.Time { return fs.modTime }
109 func (fs *fileStat) IsDir() bool      { return fs.mode.IsDir }
110 func (fs *fileStat) Sys() interface{} { return fs.sys }
111
112 // SameFile reports whether fi1 and fi2 describe the same fi
113 // For example, on Unix this means that the device and inode
114 // of the two underlying structures are identical; on other
115 // the decision may be based on the path names.
116 // SameFile only applies to results returned by this package
117 // It returns false in other cases.
118 func SameFile(fi1, fi2 FileInfo) bool {
119     fs1, ok1 := fi1.(*fileStat)
120     fs2, ok2 := fi2.(*fileStat)
121     if !ok1 || !ok2 {
122         return false
123     }
124     return sameFile(fs1.sys, fs2.sys)
125 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/exec/exec.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package exec runs external commands. It wraps os.StartPro
6 // easier to remap stdin and stdout, connect I/O with pipes,
7 // adjustments.
8 package exec
9
10 import (
11     "bytes"
12     "errors"
13     "io"
14     "os"
15     "strconv"
16     "syscall"
17 )
18
19 // Error records the name of a binary that failed to be be e
20 // and the reason it failed.
21 type Error struct {
22     Name string
23     Err  error
24 }
25
26 func (e *Error) Error() string {
27     return "exec: " + strconv.Quote(e.Name) + ": " + e.E
28 }
29
30 // Cmd represents an external command being prepared or run.
31 type Cmd struct {
32     // Path is the path of the command to run.
33     //
34     // This is the only field that must be set to a non-
35     // value.
36     Path string
37
38     // Args holds command line arguments, including the
39     // If the Args field is empty or nil, Run uses {Path
40     //
41     // In typical use, both Path and Args are set by cal
42     Args []string
43
44     // Env specifies the environment of the process.
```

```

45     // If Env is nil, Run uses the current process's env
46     Env []string
47
48     // Dir specifies the working directory of the command
49     // If Dir is the empty string, Run runs the command
50     // calling process's current directory.
51     Dir string
52
53     // Stdin specifies the process's standard input. If
54     // nil, the process reads from the null device (os.DevNull)
55     Stdin io.Reader
56
57     // Stdout and Stderr specify the process's standard
58     //
59     // If either is nil, Run connects the corresponding
60     // to the null device (os.DevNull).
61     //
62     // If Stdout and Stderr are the same writer, at most
63     // goroutine at a time will call Write.
64     Stdout io.Writer
65     Stderr io.Writer
66
67     // ExtraFiles specifies additional open files to be
68     // new process. It does not include standard input,
69     // standard error. If non-nil, entry i becomes file
70     //
71     // BUG: on OS X 10.6, child processes may sometimes
72     // http://golang.org/issue/2603
73     ExtraFiles []*os.File
74
75     // SysProcAttr holds optional, operating system-specific
76     // Run passes it to os.StartProcess as the os.ProcAttr
77     SysProcAttr *syscall.SysProcAttr
78
79     // Process is the underlying process, once started.
80     Process *os.Process
81
82     // ProcessState contains information about an exited
83     // process available after a call to Wait or Run.
84     ProcessState *os.ProcessState
85
86     err          error // last error (from LookPath,
87     finished     bool  // when Wait was called
88     childFiles   []*os.File
89     closeAfterStart []io.Closer
90     closeAfterWait []io.Closer
91     goroutine     []func() error
92     errch        chan error // one send per goroutine
93 }
94

```

```

95 // Command returns the Cmd struct to execute the named progr
96 // the given arguments.
97 //
98 // It sets Path and Args in the returned structure and zeroes
99 // other fields.
100 //
101 // If name contains no path separators, Command uses LookPath
102 // to resolve the path to a complete name if possible. Otherwise
103 // name directly.
104 //
105 // The returned Cmd's Args field is constructed from the command
106 // followed by the elements of arg, so arg should not include
107 // command name itself. For example, Command("echo", "hello")
108 func Command(name string, arg ...string) *Cmd {
109     aname, err := LookPath(name)
110     if err != nil {
111         aname = name
112     }
113     return &Cmd{
114         Path: aname,
115         Args: append([]string{name}, arg...),
116         err:  err,
117     }
118 }
119
120 // interfaceEqual protects against panics from doing equality
121 // comparisons between two interfaces with non-comparable underlying types
122 func interfaceEqual(a, b interface{}) bool {
123     defer func() {
124         recover()
125     }()
126     return a == b
127 }
128
129 func (c *Cmd) envv() []string {
130     if c.Env != nil {
131         return c.Env
132     }
133     return os.Environ()
134 }
135
136 func (c *Cmd) argv() []string {
137     if len(c.Args) > 0 {
138         return c.Args
139     }
140     return []string{c.Path}
141 }
142
143 func (c *Cmd) stdin() (f *os.File, err error) {

```

```

144     if c.Stdin == nil {
145         f, err = os.Open(os.DevNull)
146         c.closeAfterStart = append(c.closeAfterStart
147             return
148     }
149
150     if f, ok := c.Stdin.(*os.File); ok {
151         return f, nil
152     }
153
154     pr, pw, err := os.Pipe()
155     if err != nil {
156         return
157     }
158
159     c.closeAfterStart = append(c.closeAfterStart, pr)
160     c.closeAfterWait = append(c.closeAfterWait, pw)
161     c.goroutine = append(c.goroutine, func() error {
162         _, err := io.Copy(pw, c.Stdin)
163         if err1 := pw.Close(); err == nil {
164             err = err1
165         }
166         return err
167     })
168     return pr, nil
169 }
170
171 func (c *Cmd) stdout() (f *os.File, err error) {
172     return c.writerDescriptor(c.Stdout)
173 }
174
175 func (c *Cmd) stderr() (f *os.File, err error) {
176     if c.Stderr != nil && interfaceEqual(c.Stderr, c.Std
177         return c.childFiles[1], nil
178     }
179     return c.writerDescriptor(c.Stderr)
180 }
181
182 func (c *Cmd) writerDescriptor(w io.Writer) (f *os.File, err
183     if w == nil {
184         f, err = os.OpenFile(os.DevNull, os.O_WRONLY
185         c.closeAfterStart = append(c.closeAfterStart
186         return
187     }
188
189     if f, ok := w.(*os.File); ok {
190         return f, nil
191     }
192

```

```

193     pr, pw, err := os.Pipe()
194     if err != nil {
195         return
196     }
197
198     c.closeAfterStart = append(c.closeAfterStart, pw)
199     c.closeAfterWait = append(c.closeAfterWait, pr)
200     c.goroutine = append(c.goroutine, func() error {
201         _, err := io.Copy(w, pr)
202         return err
203     })
204     return pw, nil
205 }
206
207 // Run starts the specified command and waits for it to comp
208 //
209 // The returned error is nil if the command runs, has no pro
210 // copying stdin, stdout, and stderr, and exits with a zero
211 // status.
212 //
213 // If the command fails to run or doesn't complete successfu
214 // error is of type *ExitError. Other error types may be
215 // returned for I/O problems.
216 func (c *Cmd) Run() error {
217     if err := c.Start(); err != nil {
218         return err
219     }
220     return c.Wait()
221 }
222
223 // Start starts the specified command but does not wait for
224 func (c *Cmd) Start() error {
225     if c.err != nil {
226         return c.err
227     }
228     if c.Process != nil {
229         return errors.New("exec: already started")
230     }
231
232     type F func(*Cmd) (*os.File, error)
233     for _, setupFd := range []F{(*Cmd).stdin, (*Cmd).std
234         fd, err := setupFd(c)
235         if err != nil {
236             return err
237         }
238         c.childFiles = append(c.childFiles, fd)
239     }
240     c.childFiles = append(c.childFiles, c.ExtraFiles...)
241
242     var err error

```

```

243         c.Process, err = os.StartProcess(c.Path, c.argv(), &
244             Dir:    c.Dir,
245             Files:  c.childFiles,
246             Env:    c.envv(),
247             Sys:    c.SysProcAttr,
248         })
249         if err != nil {
250             return err
251         }
252
253         for _, fd := range c.closeAfterStart {
254             fd.Close()
255         }
256
257         c.errch = make(chan error, len(c.goroutine))
258         for _, fn := range c.goroutine {
259             go func(fn func() error) {
260                 c.errch <- fn()
261             }(fn)
262         }
263
264         return nil
265     }
266
267     // An ExitError reports an unsuccessful exit by a command.
268     type ExitError struct {
269         *os.ProcessState
270     }
271
272     func (e *ExitError) Error() string {
273         return e.ProcessState.String()
274     }
275
276     // Wait waits for the command to exit.
277     // It must have been started by Start.
278     //
279     // The returned error is nil if the command runs, has no pro
280     // copying stdin, stdout, and stderr, and exits with a zero
281     // status.
282     //
283     // If the command fails to run or doesn't complete successfu
284     // error is of type *ExitError. Other error types may be
285     // returned for I/O problems.
286     func (c *Cmd) Wait() error {
287         if c.Process == nil {
288             return errors.New("exec: not started")
289         }
290         if c.finished {
291             return errors.New("exec: Wait was already ca

```

```

292     }
293     c.finished = true
294     state, err := c.Process.Wait()
295     c.ProcessState = state
296
297     var copyError error
298     for _ = range c.goroutine {
299         if err := <-c.errch; err != nil && copyError
300             copyError = err
301     }
302 }
303
304     for _, fd := range c.closeAfterWait {
305         fd.Close()
306     }
307
308     if err != nil {
309         return err
310     } else if !state.Success() {
311         return &ExitError{state}
312     }
313
314     return copyError
315 }
316
317 // Output runs the command and returns its standard output.
318 func (c *Cmd) Output() ([]byte, error) {
319     if c.Stdout != nil {
320         return nil, errors.New("exec: Stdout already
321     }
322     var b bytes.Buffer
323     c.Stdout = &b
324     err := c.Run()
325     return b.Bytes(), err
326 }
327
328 // CombinedOutput runs the command and returns its combined
329 // output and standard error.
330 func (c *Cmd) CombinedOutput() ([]byte, error) {
331     if c.Stdout != nil {
332         return nil, errors.New("exec: Stdout already
333     }
334     if c.Stderr != nil {
335         return nil, errors.New("exec: Stderr already
336     }
337     var b bytes.Buffer
338     c.Stdout = &b
339     c.Stderr = &b
340     err := c.Run()

```

```

341         return b.Bytes(), err
342     }
343
344     // StdinPipe returns a pipe that will be connected to the co
345     // standard input when the command starts.
346     func (c *Cmd) StdinPipe() (io.WriteCloser, error) {
347         if c.Stdin != nil {
348             return nil, errors.New("exec: Stdin already
349         }
350         if c.Process != nil {
351             return nil, errors.New("exec: StdinPipe afte
352         }
353         pr, pw, err := os.Pipe()
354         if err != nil {
355             return nil, err
356         }
357         c.Stdin = pr
358         c.closeAfterStart = append(c.closeAfterStart, pr)
359         c.closeAfterWait = append(c.closeAfterWait, pw)
360         return pw, nil
361     }
362
363     // StdoutPipe returns a pipe that will be connected to the c
364     // standard output when the command starts.
365     // The pipe will be closed automatically after Wait sees the
366     func (c *Cmd) StdoutPipe() (io.ReadCloser, error) {
367         if c.Stdout != nil {
368             return nil, errors.New("exec: Stdout already
369         }
370         if c.Process != nil {
371             return nil, errors.New("exec: StdoutPipe aft
372         }
373         pr, pw, err := os.Pipe()
374         if err != nil {
375             return nil, err
376         }
377         c.Stdout = pw
378         c.closeAfterStart = append(c.closeAfterStart, pw)
379         c.closeAfterWait = append(c.closeAfterWait, pr)
380         return pr, nil
381     }
382
383     // StderrPipe returns a pipe that will be connected to the c
384     // standard error when the command starts.
385     // The pipe will be closed automatically after Wait sees the
386     func (c *Cmd) StderrPipe() (io.ReadCloser, error) {
387         if c.Stderr != nil {
388             return nil, errors.New("exec: Stderr already
389         }
390         if c.Process != nil {

```

```
391         return nil, errors.New("exec: StderrPipe aft
392     }
393     pr, pw, err := os.Pipe()
394     if err != nil {
395         return nil, err
396     }
397     c.Stderr = pw
398     c.closeAfterStart = append(c.closeAfterStart, pw)
399     c.closeAfterWait = append(c.closeAfterWait, pr)
400     return pr, nil
401 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/exec/lp_unix.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package exec
8
9 import (
10     "errors"
11     "os"
12     "strings"
13 )
14
15 // ErrNotFound is the error resulting if a path search failed
16 var ErrNotFound = errors.New("executable file not found in $
17
18 func findExecutable(file string) error {
19     d, err := os.Stat(file)
20     if err != nil {
21         return err
22     }
23     if m := d.Mode(); !m.IsDir() && m&0111 != 0 {
24         return nil
25     }
26     return os.ErrPermission
27 }
28
29 // LookPath searches for an executable binary named file
30 // in the directories named by the PATH environment variable
31 // If file contains a slash, it is tried directly and the PA
32 func LookPath(file string) (string, error) {
33     // NOTE(rsc): I wish we could use the Plan 9 behavior
34     // (only bypass the path if file begins with / or ./
35     // but that would not match all the Unix shells.
36
37     if strings.Contains(file, "/") {
38         err := findExecutable(file)
39         if err == nil {
40             return file, nil
41         }
42         return "", &Error{file, err}
43     }
44     pathenv := os.Getenv("PATH")
```

```
45     for _, dir := range strings.Split(pathenv, ":") {
46         if dir == "" {
47             // Unix shell semantics: path element
48             dir = "."
49         }
50         path := dir + "/" + file
51         if err := findExecutable(path); err == nil {
52             return path, nil
53         }
54     }
55     return "", &Error{file, ErrNotFound}
56 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/signal/signal.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package signal implements access to incoming signals.
6 package signal
7
8 // BUG(rsc): This package is not yet implemented on Plan 9 a
9
10 import (
11     "os"
12     "sync"
13 )
14
15 var handlers struct {
16     sync.Mutex
17     list []handler
18 }
19
20 type handler struct {
21     c chan<- os.Signal
22     sig os.Signal
23     all bool
24 }
25
26 // Notify causes package signal to relay incoming signals to
27 // If no signals are listed, all incoming signals will be re
28 // Otherwise, just the listed signals will.
29 //
30 // Package signal will not block sending to c: the caller mu
31 // that c has sufficient buffer space to keep up with the ex
32 // signal rate. For a channel used for notification of just
33 // a buffer of size 1 is sufficient.
34 //
35 func Notify(c chan<- os.Signal, sig ...os.Signal) {
36     if c == nil {
37         panic("os/signal: Notify using nil channel")
38     }
39
40     handlers.Lock()
41     defer handlers.Unlock()
42     if len(sig) == 0 {
43         enableSignal(nil)
44         handlers.list = append(handlers.list, handle
```

```

45     } else {
46         for _, s := range sig {
47             // We use nil as a special wildcard
48             // so filter it out of the list of a
49             // we will never get an incoming nil
50             // registration cannot affect the ob
51             if s != nil {
52                 enableSignal(s)
53                 handlers.list = append(handl
54             }
55         }
56     }
57 }
58
59 func process(sig os.Signal) {
60     handlers.Lock()
61     defer handlers.Unlock()
62
63     for _, h := range handlers.list {
64         if h.all || h.sig == sig {
65             // send but do not block for it
66             select {
67                 case h.c <- sig:
68                 default:
69             }
70         }
71     }
72 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/os/signal/signal_unix.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd windows
6
7 package signal
8
9 import (
10     "os"
11     "syscall"
12 )
13
14 // In assembly.
15 func signal_enable(uint32)
16 func signal_recv() uint32
17
18 func loop() {
19     for {
20         process(syscall.Signal(signal_recv()))
21     }
22 }
23
24 func init() {
25     signal_enable(0) // first call - initialize
26     go loop()
27 }
28
29 func enableSignal(sig os.Signal) {
30     switch sig := sig.(type) {
31     case nil:
32         signal_enable(^uint32(0))
33     case syscall.Signal:
34         signal_enable(uint32(sig))
35     default:
36         // Can ignore: this signal (whatever it is)
37     }
38 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/os/user/lookup_unix.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux
6 // +build cgo
7
8 package user
9
10 import (
11     "fmt"
12     "runtime"
13     "strconv"
14     "strings"
15     "syscall"
16     "unsafe"
17 )
18
19 /*
20 #include <unistd.h>
21 #include <sys/types.h>
22 #include <pwd.h>
23 #include <stdlib.h>
24
25 static int mygetpwuid_r(int uid, struct passwd *pwd,
26     char *buf, size_t buflen, struct passwd **result) {
27     return getpwuid_r(uid, pwd, buf, buflen, result);
28 }
29 */
30 import "C"
31
32 // Current returns the current user.
33 func Current() (*User, error) {
34     return lookup(syscall.Getuid(), "", false)
35 }
36
37 // Lookup looks up a user by username. If the user cannot be
38 // the returned error is of type UnknownUserError.
39 func Lookup(username string) (*User, error) {
40     return lookup(-1, username, true)
41 }
```

```

42
43 // LookupId looks up a user by userid. If the user cannot be
44 // the returned error is of type UnknownUserIdError.
45 func LookupId(uid string) (*User, error) {
46     i, e := strconv.Atoi(uid)
47     if e != nil {
48         return nil, e
49     }
50     return lookup(i, "", false)
51 }
52
53 func lookup(uid int, username string, lookupByName bool) (*U
54     var pwd C.struct_passwd
55     var result *C.struct_passwd
56
57     var bufSize C.long
58     if runtime.GOOS == "freebsd" {
59         // FreeBSD doesn't have _SC_GETPW_R_SIZE_MAX
60         // and just returns -1. So just use the sam
61         // size that Linux returns
62         bufSize = 1024
63     } else {
64         bufSize = C.sysconf(C._SC_GETPW_R_SIZE_MAX)
65         if bufSize <= 0 || bufSize > 1<<20 {
66             return nil, fmt.Errorf("user: unreas
67         }
68     }
69     buf := C.malloc(C.size_t(bufSize))
70     defer C.free(buf)
71     var rv C.int
72     if lookupByName {
73         nameC := C.CString(username)
74         defer C.free(unsafe.Pointer(nameC))
75         rv = C.getpwnam_r(nameC,
76             &pwd,
77             (*C.char)(buf),
78             C.size_t(bufSize),
79             &result)
80         if rv != 0 {
81             return nil, fmt.Errorf("user: lookup
82         }
83         if result == nil {
84             return nil, UnknownUserError(usernam
85         }
86     } else {
87         // mygetpwuid_r is a wrapper around getpwuid
88         // to avoid using uid_t because C.uid_t(uid)
89         // unknown reasons doesn't work on linux.
90         rv = C.mygetpwuid_r(C.int(uid),
91             &pwd,

```

```

92         (*C.char)(buf),
93         C.size_t(bufSize),
94         &result)
95     if rv != 0 {
96         return nil, fmt.Errorf("user: lookup
97     }
98     if result == nil {
99         return nil, UnknownUserIdError(uid)
100    }
101 }
102 u := &User{
103     Uid:      strconv.Itoa(int(pwd.pw_uid)),
104     Gid:      strconv.Itoa(int(pwd.pw_gid)),
105     Username: C.GoString(pwd.pw_name),
106     Name:     C.GoString(pwd.pw_gecos),
107     HomeDir:  C.GoString(pwd.pw_dir),
108 }
109 // The pw_gecos field isn't quite standardized. Some
110 // say: "It is expected to be a comma separated list
111 // of personal data where the first item is the full name
112 // of the user."
113 if i := strings.Index(u.Name, ","); i >= 0 {
114     u.Name = u.Name[:i]
115 }
116 return u, nil
117 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/os/user/user.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package user allows user account lookups by name or id.
6 package user
7
8 import (
9     "strconv"
10 )
11
12 var implemented = true // set to false by lookup_stubs.go's
13
14 // User represents a user account.
15 //
16 // On posix systems Uid and Gid contain a decimal number
17 // representing uid and gid. On windows Uid and Gid
18 // contain security identifier (SID) in a string format.
19 type User struct {
20     Uid      string // user id
21     Gid      string // primary group id
22     Username string
23     Name     string
24     HomeDir  string
25 }
26
27 // UnknownUserIdError is returned by LookupId when
28 // a user cannot be found.
29 type UnknownUserIdError int
30
31 func (e UnknownUserIdError) Error() string {
32     return "user: unknown userid " + strconv.Itoa(int(e))
33 }
34
35 // UnknownUserError is returned by Lookup when
36 // a user cannot be found.
37 type UnknownUserError string
38
39 func (e UnknownUserError) Error() string {
40     return "user: unknown user " + string(e)
41 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/path/match.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package path
6
7 import (
8     "errors"
9     "strings"
10    "unicode/utf8"
11 )
12
13 // ErrBadPattern indicates a globbing pattern was malformed.
14 var ErrBadPattern = errors.New("syntax error in pattern")
15
16 // Match returns true if name matches the shell file name pa
17 // The pattern syntax is:
18 //
19 //     pattern:
20 //         { term }
21 //     term:
22 //         '*'           matches any sequence of non-/ ch
23 //         '?'          matches any single non-/ charact
24 //         '[' [ '^' ] { character-range } ']'
25 //                   character class (must be non-emp
26 //         c             matches character c (c != '*', '
27 //         '\\' c       matches character c
28 //
29 //     character-range:
30 //         c             matches character c (c != '\\',
31 //         '\\' c       matches character c
32 //         lo '-' hi    matches character c for lo <= c
33 //
34 // Match requires pattern to match all of name, not just a s
35 // The only possible returned error is ErrBadPattern, when p
36 // is malformed.
37 //
38 func Match(pattern, name string) (matched bool, err error) {
39     Pattern:
40         for len(pattern) > 0 {
41             var star bool
42             var chunk string
43             star, chunk, pattern = scanChunk(pattern)
44             if star && chunk == "" {
```

```

45         // Trailing * matches rest of string
46         return strings.Index(name, "/") < 0,
47     }
48     // Look for match at current position.
49     t, ok, err := matchChunk(chunk, name)
50     // if we're the last chunk, make sure we've
51     // otherwise we'll give a false result even
52     // using the star
53     if ok && (len(t) == 0 || len(pattern) > 0) {
54         name = t
55         continue
56     }
57     if err != nil {
58         return false, err
59     }
60     if star {
61         // Look for match skipping i+1 bytes
62         // Cannot skip /.
63         for i := 0; i < len(name) && name[i]
64             t, ok, err := matchChunk(chu
65             if ok {
66                 // if we're the last
67                 if len(pattern) == 0
68                     continue
69             }
70             name = t
71             continue Pattern
72         }
73         if err != nil {
74             return false, err
75         }
76     }
77     }
78     return false, nil
79 }
80 return len(name) == 0, nil
81 }
82
83 // scanChunk gets the next segment of pattern, which is a no
84 // possibly preceded by a star.
85 func scanChunk(pattern string) (star bool, chunk, rest strin
86     for len(pattern) > 0 && pattern[0] == '*' {
87         pattern = pattern[1:]
88         star = true
89     }
90     inrange := false
91     var i int
92 Scan:
93     for i = 0; i < len(pattern); i++ {
94         switch pattern[i] {

```

```

95         case '\\':
96             // error check handled in matchChunk
97             if i+1 < len(pattern) {
98                 i++
99             }
100        case '[':
101            inrange = true
102        case ']':
103            inrange = false
104        case '*':
105            if !inrange {
106                break Scan
107            }
108        }
109    }
110    return star, pattern[0:i], pattern[i:]
111 }
112
113 // matchChunk checks whether chunk matches the beginning of
114 // If so, it returns the remainder of s (after the match).
115 // Chunk is all single-character operators: literals, char c
116 func matchChunk(chunk, s string) (rest string, ok bool, err
117     for len(chunk) > 0 {
118         if len(s) == 0 {
119             return
120         }
121         switch chunk[0] {
122         case '[':
123             // character class
124             r, n := utf8.DecodeRuneInString(s)
125             s = s[n:]
126             chunk = chunk[1:]
127             // possibly negated
128             notNegated := true
129             if len(chunk) > 0 && chunk[0] == '^'
130                 notNegated = false
131                 chunk = chunk[1:]
132             }
133             // parse all ranges
134             match := false
135             nrange := 0
136             for {
137                 if len(chunk) > 0 && chunk[0]
138                     chunk = chunk[1:]
139                     break
140             }
141             var lo, hi rune
142             if lo, chunk, err = getEsc(c
143                 return

```

```

144     }
145     hi = lo
146     if chunk[0] == '-' {
147         if hi, chunk, err =
148             return
149     }
150     }
151     if lo <= r && r <= hi {
152         match = true
153     }
154     nrange++
155 }
156 if match != notNegated {
157     return
158 }
159
160 case '?':
161     if s[0] == '/' {
162         return
163     }
164     _, n := utf8.DecodeRuneInString(s)
165     s = s[n:]
166     chunk = chunk[1:]
167
168 case '\\':
169     chunk = chunk[1:]
170     if len(chunk) == 0 {
171         err = ErrBadPattern
172         return
173     }
174     fallthrough
175
176 default:
177     if chunk[0] != s[0] {
178         return
179     }
180     s = s[1:]
181     chunk = chunk[1:]
182 }
183 }
184 return s, true, nil
185 }
186
187 // getEsc gets a possibly-escaped character from chunk, for
188 func getEsc(chunk string) (r rune, nchunk string, err error)
189     if len(chunk) == 0 || chunk[0] == '-' || chunk[0] ==
190         err = ErrBadPattern
191         return
192     }

```

```
193     if chunk[0] == '\\\\' {
194         chunk = chunk[1:]
195         if len(chunk) == 0 {
196             err = ErrBadPattern
197             return
198         }
199     }
200     r, n := utf8.DecodeRuneInString(chunk)
201     if r == utf8.RuneError && n == 1 {
202         err = ErrBadPattern
203     }
204     nchunk = chunk[n:]
205     if len(nchunk) == 0 {
206         err = ErrBadPattern
207     }
208     return
209 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/path/path.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package path implements utility routines for manipulating
6 // paths.
7 package path
8
9 import (
10     "strings"
11 )
12
13 // Clean returns the shortest path name equivalent to path
14 // by purely lexical processing. It applies the following r
15 // iteratively until no further processing can be done:
16 //
17 //     1. Replace multiple slashes with a single slash.
18 //     2. Eliminate each . path name element (the current d
19 //     3. Eliminate each inner .. path name element (the pa
20 //        along with the non-.. element that precedes it.
21 //     4. Eliminate .. elements that begin a rooted path:
22 //        that is, replace "/.." by "/" at the beginning of
23 //
24 // The returned path ends in a slash only if it is the root
25 //
26 // If the result of this process is an empty string, Clean
27 // returns the string ".".
28 //
29 // See also Rob Pike, ``Lexical File Names in Plan 9 or
30 // Getting Dot-Dot Right,''
31 // http://plan9.bell-labs.com/sys/doc/lexnames.html
32 func Clean(path string) string {
33     if path == "" {
34         return "."
35     }
36
37     rooted := path[0] == '/'
38     n := len(path)
39
40     // Invariants:
41     //     reading from path; r is index of next byte t
42     //     writing to buf; w is index of next byte to w
43     //     dotdot is index in buf where .. must stop, e
44     //     it is the leading slash or it is a l
```

```

45     buf := []byte(path)
46     r, w, dotdot := 0, 0, 0
47     if rooted {
48         r, w, dotdot = 1, 1, 1
49     }
50
51     for r < n {
52         switch {
53         case path[r] == '/':
54             // empty path element
55             r++
56         case path[r] == '.' && (r+1 == n || path[r+1]
57             // . element
58             r++
59         case path[r] == '.' && path[r+1] == '.' && (
60             // .. element: remove to last /
61             r += 2
62             switch {
63             case w > dotdot:
64                 // can backtrack
65                 w--
66                 for w > dotdot && buf[w] !=
67                     w--
68             }
69             case !rooted:
70                 // cannot backtrack, but not
71                 if w > 0 {
72                     buf[w] = '/'
73                     w++
74                 }
75                 buf[w] = '.'
76                 w++
77                 buf[w] = '.'
78                 w++
79                 dotdot = w
80             }
81         default:
82             // real path element.
83             // add slash if needed
84             if rooted && w != 1 || !rooted && w
85                 buf[w] = '/'
86                 w++
87             }
88             // copy element
89             for ; r < n && path[r] != '/'; r++ {
90                 buf[w] = path[r]
91                 w++
92             }
93         }
94     }

```

```

95
96     // Turn empty string into "."
97     if w == 0 {
98         buf[w] = '.'
99         w++
100    }
101
102    return string(buf[0:w])
103 }
104
105 // Split splits path immediately following the final slash.
106 // separating it into a directory and file name component.
107 // If there is no slash path, Split returns an empty dir and
108 // file set to path.
109 // The returned values have the property that path = dir+file
110 func Split(path string) (dir, file string) {
111     i := strings.LastIndex(path, "/")
112     return path[:i+1], path[i+1:]
113 }
114
115 // Join joins any number of path elements into a single path
116 // separating slash if necessary. The result is Cleaned; in
117 // all empty strings are ignored.
118 func Join(elem ...string) string {
119     for i, e := range elem {
120         if e != "" {
121             return Clean(strings.Join(elem[i:],
122                                     "/"))
123         }
124     }
125     return ""
126 }
127 // Ext returns the file name extension used by path.
128 // The extension is the suffix beginning at the final dot
129 // in the final slash-separated element of path;
130 // it is empty if there is no dot.
131 func Ext(path string) string {
132     for i := len(path) - 1; i >= 0 && path[i] != '/'; i-- {
133         if path[i] == '.' {
134             return path[i:]
135         }
136     }
137     return ""
138 }
139
140 // Base returns the last element of path.
141 // Trailing slashes are removed before extracting the last e
142 // If the path is empty, Base returns ".".
143 // If the path consists entirely of slashes, Base returns "/"

```

```

144 func Base(path string) string {
145     if path == "" {
146         return "."
147     }
148     // Strip trailing slashes.
149     for len(path) > 0 && path[len(path)-1] == '/' {
150         path = path[0 : len(path)-1]
151     }
152     // Find the last element
153     if i := strings.LastIndex(path, "/"); i >= 0 {
154         path = path[i+1:]
155     }
156     // If empty now, it had only slashes.
157     if path == "" {
158         return "/"
159     }
160     return path
161 }
162
163 // IsAbs returns true if the path is absolute.
164 func IsAbs(path string) bool {
165     return len(path) > 0 && path[0] == '/'
166 }
167
168 // Dir returns all but the last element of path, typically t
169 // The path is Cleaned and trailing slashes are removed befo
170 // If the path is empty, Dir returns ".".
171 // If the path consists entirely of slashes followed by non-
172 // returns a single slash. In any other case, the returned p
173 // slash.
174 func Dir(path string) string {
175     dir, _ := Split(path)
176     dir = Clean(dir)
177     last := len(dir) - 1
178     if last > 0 && dir[last] == '/' {
179         dir = dir[:last]
180     }
181     if dir == "" {
182         dir = "."
183     }
184     return dir
185 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/path/filepath/match.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package filepath
6
7 import (
8     "errors"
9     "os"
10    "runtime"
11    "sort"
12    "strings"
13    "unicode/utf8"
14 )
15
16 // ErrBadPattern indicates a globbing pattern was malformed.
17 var ErrBadPattern = errors.New("syntax error in pattern")
18
19 // Match returns true if name matches the shell file name pa
20 // The pattern syntax is:
21 //
22 //     pattern:
23 //         { term }
24 //     term:
25 //         '*'           matches any sequence of non-Sepa
26 //         '?'           matches any single non-Separator
27 //         '[' [ '^' ] { character-range } ']'
28 //                     character class (must be non-emp
29 //         c             matches character c (c != '*', '
30 //         '\\' c       matches character c
31 //
32 //     character-range:
33 //         c             matches character c (c != '\\',
34 //         '\\' c       matches character c
35 //         lo '-' hi    matches character c for lo <= c
36 //
37 // Match requires pattern to match all of name, not just a s
38 // The only possible returned error is ErrBadPattern, when p
39 // is malformed.
40 //
41 // On Windows, escaping is disabled. Instead, '\\' is treate
```

```

42 // path separator.
43 //
44 func Match(pattern, name string) (matched bool, err error) {
45 Pattern:
46     for len(pattern) > 0 {
47         var star bool
48         var chunk string
49         star, chunk, pattern = scanChunk(pattern)
50         if star && chunk == "" {
51             // Trailing * matches rest of string
52             return strings.Index(name, string(Se
53         }
54         // Look for match at current position.
55         t, ok, err := matchChunk(chunk, name)
56         // if we're the last chunk, make sure we've
57         // otherwise we'll give a false result even
58         // using the star
59         if ok && (len(t) == 0 || len(pattern) > 0) {
60             name = t
61             continue
62         }
63         if err != nil {
64             return false, err
65         }
66         if star {
67             // Look for match skipping i+1 bytes
68             // Cannot skip /.
69             for i := 0; i < len(name) && name[i]
70                 t, ok, err := matchChunk(chu
71                 if ok {
72                     // if we're the last
73                     if len(pattern) == 0
74                         continue
75                 }
76                 name = t
77                 continue Pattern
78             }
79             if err != nil {
80                 return false, err
81             }
82         }
83     }
84     return false, nil
85 }
86 return len(name) == 0, nil
87 }
88
89 // scanChunk gets the next segment of pattern, which is a no
90 // possibly preceded by a star.
91 func scanChunk(pattern string) (star bool, chunk, rest strin

```

```

92     for len(pattern) > 0 && pattern[0] == '*' {
93         pattern = pattern[1:]
94         star = true
95     }
96     inrange := false
97     var i int
98     Scan:
99     for i = 0; i < len(pattern); i++ {
100         switch pattern[i] {
101             case '\\':
102                 if runtime.GOOS != "windows" {
103                     // error check handled in ma
104                     if i+1 < len(pattern) {
105                         i++
106                     }
107                 }
108             case '[':
109                 inrange = true
110             case ']':
111                 inrange = false
112             case '*':
113                 if !inrange {
114                     break Scan
115                 }
116             }
117         }
118     return star, pattern[0:i], pattern[i:]
119 }
120
121 // matchChunk checks whether chunk matches the beginning of
122 // If so, it returns the remainder of s (after the match).
123 // Chunk is all single-character operators: literals, char c
124 func matchChunk(chunk, s string) (rest string, ok bool, err
125     for len(chunk) > 0 {
126         if len(s) == 0 {
127             return
128         }
129         switch chunk[0] {
130             case '[':
131                 // character class
132                 r, n := utf8.DecodeRuneInString(s)
133                 s = s[n:]
134                 chunk = chunk[1:]
135                 // possibly negated
136                 negated := chunk[0] == '^'
137                 if negated {
138                     chunk = chunk[1:]
139                 }
140                 // parse all ranges

```

```

141         match := false
142         nrange := 0
143         for {
144             if len(chunk) > 0 && chunk[0]
145                 chunk = chunk[1:]
146                 break
147         }
148         var lo, hi rune
149         if lo, chunk, err = getEsc(c)
150             return
151         }
152         hi = lo
153         if chunk[0] == '-' {
154             if hi, chunk, err =
155                 return
156             }
157         }
158         if lo <= r && r <= hi {
159             match = true
160         }
161         nrange++
162     }
163     if match == negated {
164         return
165     }
166
167     case '?':
168         if s[0] == Separator {
169             return
170         }
171         _, n := utf8.DecodeRuneInString(s)
172         s = s[n:]
173         chunk = chunk[1:]
174
175     case '\\':
176         if runtime.GOOS != "windows" {
177             chunk = chunk[1:]
178             if len(chunk) == 0 {
179                 err = ErrBadPattern
180                 return
181             }
182         }
183         fallthrough
184
185     default:
186         if chunk[0] != s[0] {
187             return
188         }
189         s = s[1:]

```

```

190             chunk = chunk[1:]
191         }
192     }
193     return s, true, nil
194 }
195
196 // getEsc gets a possibly-escaped character from chunk, for
197 func getEsc(chunk string) (r rune, nchunk string, err error)
198     if len(chunk) == 0 || chunk[0] == '-' || chunk[0] ==
199         err = ErrBadPattern
200         return
201     }
202     if chunk[0] == '\\' && runtime.GOOS != "windows" {
203         chunk = chunk[1:]
204         if len(chunk) == 0 {
205             err = ErrBadPattern
206             return
207         }
208     }
209     r, n := utf8.DecodeRuneInString(chunk)
210     if r == utf8.RuneError && n == 1 {
211         err = ErrBadPattern
212     }
213     nchunk = chunk[n:]
214     if len(nchunk) == 0 {
215         err = ErrBadPattern
216     }
217     return
218 }
219
220 // Glob returns the names of all files matching pattern or n
221 // if there is no matching file. The syntax of patterns is t
222 // as in Match. The pattern may describe hierarchical names
223 // /usr/*/bin/ed (assuming the Separator is '/').
224 //
225 func Glob(pattern string) (matches []string, err error) {
226     if !hasMeta(pattern) {
227         if _, err = os.Stat(pattern); err != nil {
228             return nil, nil
229         }
230         return []string{pattern}, nil
231     }
232
233     dir, file := Split(pattern)
234     switch dir {
235     case "":
236         dir = "."
237     case string(Separator):
238         // nothing
239     default:

```

```

240         dir = dir[0 : len(dir)-1] // chop off traili
241     }
242
243     if !hasMeta(dir) {
244         return glob(dir, file, nil)
245     }
246
247     var m []string
248     m, err = Glob(dir)
249     if err != nil {
250         return
251     }
252     for _, d := range m {
253         matches, err = glob(d, file, matches)
254         if err != nil {
255             return
256         }
257     }
258     return
259 }
260
261 // glob searches for files matching pattern in the directory
262 // and appends them to matches. If the directory cannot be
263 // opened, it returns the existing matches. New matches are
264 // added in lexicographical order.
265 func glob(dir, pattern string, matches []string) (m []string)
266     m = matches
267     fi, err := os.Stat(dir)
268     if err != nil {
269         return
270     }
271     if !fi.IsDir() {
272         return
273     }
274     d, err := os.Open(dir)
275     if err != nil {
276         return
277     }
278     defer d.Close()
279
280     names, err := d.Readdirnames(-1)
281     if err != nil {
282         return
283     }
284     sort.Strings(names)
285
286     for _, n := range names {
287         matched, err := Match(pattern, n)
288         if err != nil {

```

```
289             return m, err
290         }
291         if matched {
292             m = append(m, Join(dir, n))
293         }
294     }
295     return
296 }
297
298 // hasMeta returns true if path contains any of the magic ch
299 // recognized by Match.
300 func hasMeta(path string) bool {
301     // TODO(niemeyer): Should other magic characters be
302     return strings.IndexAny(path, "*?[") >= 0
303 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/path/filepath/path.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package filepath implements utility routines for manipula
6 // in a way compatible with the target operating system-defi
7 package filepath
8
9 import (
10     "errors"
11     "os"
12     "sort"
13     "strings"
14 )
15
16 const (
17     Separator      = os.PathSeparator
18     ListSeparator  = os.PathListSeparator
19 )
20
21 // Clean returns the shortest path name equivalent to path
22 // by purely lexical processing. It applies the following r
23 // iteratively until no further processing can be done:
24 //
25 //     1. Replace multiple Separator elements with a single
26 //     2. Eliminate each . path name element (the current d
27 //     3. Eliminate each inner .. path name element (the pa
28 //     along with the non-.. element that precedes it.
29 //     4. Eliminate .. elements that begin a rooted path:
30 //     that is, replace "/.." by "/" at the beginning of
31 //     assuming Separator is '/'.
32 //
33 // The returned path ends in a slash only if it represents a
34 // such as "/" on Unix or `C:\` on Windows.
35 //
36 // If the result of this process is an empty string, Clean
37 // returns the string ".".
38 //
39 // See also Rob Pike, ``Lexical File Names in Plan 9 or
40 // Getting Dot-Dot Right, ''
41 // http://plan9.bell-labs.com/sys/doc/lexnames.html
```

```

42 func Clean(path string) string {
43     vol := VolumeName(path)
44     path = path[len(vol):]
45     if path == "" {
46         if len(vol) > 1 && vol[1] != ':' {
47             // should be UNC
48             return FromSlash(vol)
49         }
50         return vol + "."
51     }
52     rooted := os.IsPathSeparator(path[0])
53
54     // Invariants:
55     //     reading from path; r is index of next byte t
56     //     writing to buf; w is index of next byte to w
57     //     dotdot is index in buf where .. must stop, e
58     //     it is the leading slash or it is a l
59     n := len(path)
60     buf := []byte(path)
61     r, w, dotdot := 0, 0, 0
62     if rooted {
63         buf[0] = Separator
64         r, w, dotdot = 1, 1, 1
65     }
66
67     for r < n {
68         switch {
69             case os.IsPathSeparator(path[r]):
70                 // empty path element
71                 r++
72             case path[r] == '.' && (r+1 == n || os.IsPat
73                 // . element
74                 r++
75             case path[r] == '.' && path[r+1] == '.' && (
76                 // .. element: remove to last separa
77                 r += 2
78                 switch {
79                     case w > dotdot:
80                         // can backtrack
81                         w--
82                         for w > dotdot && !os.IsPath
83                             w--
84                 }
85             case !rooted:
86                 // cannot backtrack, but not
87                 if w > 0 {
88                     buf[w] = Separator
89                     w++
90                 }
91                 buf[w] = '.'

```

```

92             w++
93             buf[w] = '.'
94             w++
95             dotdot = w
96         }
97     default:
98         // real path element.
99         // add slash if needed
100        if rooted && w != 1 || !rooted && w
101            buf[w] = Separator
102            w++
103        }
104        // copy element
105        for ; r < n && !os.IsPathSeparator(p
106            buf[w] = path[r]
107            w++
108        }
109    }
110 }
111
112 // Turn empty string into "."
113 if w == 0 {
114     buf[w] = '.'
115     w++
116 }
117
118 return FromSlash(vol + string(buf[0:w]))
119 }
120
121 // ToSlash returns the result of replacing each separator ch
122 // in path with a slash ('/') character. Multiple separators
123 // replaced by multiple slashes.
124 func ToSlash(path string) string {
125     if Separator == '/' {
126         return path
127     }
128     return strings.Replace(path, string(Separator), "/",
129 }
130
131 // FromSlash returns the result of replacing each slash ('/'
132 // in path with a separator character. Multiple slashes are
133 // by multiple separators.
134 func FromSlash(path string) string {
135     if Separator == '/' {
136         return path
137     }
138     return strings.Replace(path, "/", string(Separator),
139 }
140

```

```

141 // SplitList splits a list of paths joined by the OS-specific
142 // usually found in PATH or GOPATH environment variables.
143 // Unlike strings.Split, SplitList returns an empty slice wh
144 func SplitList(path string) []string {
145     if path == "" {
146         return []string{}
147     }
148     return strings.Split(path, string(ListSeparator))
149 }
150
151 // Split splits path immediately following the final Separator
152 // separating it into a directory and file name component.
153 // If there is no Separator in path, Split returns an empty
154 // and file set to path.
155 // The returned values have the property that path = dir+file
156 func Split(path string) (dir, file string) {
157     vol := VolumeName(path)
158     i := len(path) - 1
159     for i >= len(vol) && !os.IsPathSeparator(path[i]) {
160         i--
161     }
162     return path[:i+1], path[i+1:]
163 }
164
165 // Join joins any number of path elements into a single path
166 // a Separator if necessary. The result is Cleaned, in parti
167 // all empty strings are ignored.
168 func Join(elem ...string) string {
169     for i, e := range elem {
170         if e != "" {
171             return Clean(strings.Join(elem[i:],
172                                     string(ListSeparator)))
173         }
174     }
175     return ""
176 }
177
178 // Ext returns the file name extension used by path.
179 // The extension is the suffix beginning at the final dot
180 // in the final element of path; it is empty if there is
181 // no dot.
182 func Ext(path string) string {
183     for i := len(path) - 1; i >= 0 && !os.IsPathSeparato
184         if path[i] == '.' {
185             return path[i:]
186         }
187     }
188     return ""
189 }

```

```

190 // EvalSymlinks returns the path name after the evaluation o
191 // links.
192 // If path is relative the result will be relative to the cu
193 // unless one of the components is an absolute symbolic link
194 func EvalSymlinks(path string) (string, error) {
195     return evalSymlinks(path)
196 }
197
198 // Abs returns an absolute representation of path.
199 // If the path is not absolute it will be joined with the cu
200 // working directory to turn it into an absolute path. The
201 // path name for a given file is not guaranteed to be unique
202 func Abs(path string) (string, error) {
203     if IsAbs(path) {
204         return Clean(path), nil
205     }
206     wd, err := os.Getwd()
207     if err != nil {
208         return "", err
209     }
210     return Join(wd, path), nil
211 }
212
213 // Rel returns a relative path that is lexically equivalent
214 // joined to basepath with an intervening separator. That is
215 // Join(basepath, Rel(basepath, targpath)) is equivalent to
216 // On success, the returned path will always be relative to
217 // even if basepath and targpath share no elements.
218 // An error is returned if targpath can't be made relative t
219 // knowing the current working directory would be necessary
220 func Rel(basepath, targpath string) (string, error) {
221     baseVol := VolumeName(basepath)
222     targVol := VolumeName(targpath)
223     base := Clean(basepath)
224     targ := Clean(targpath)
225     if targ == base {
226         return ".", nil
227     }
228     base = base[len(baseVol):]
229     targ = targ[len(targVol):]
230     if base == "." {
231         base = ""
232     }
233     // Can't use IsAbs - `a` and `a` are both relative
234     baseSlashed := len(base) > 0 && base[0] == Separator
235     targSlashed := len(targ) > 0 && targ[0] == Separator
236     if baseSlashed != targSlashed || baseVol != targVol
237         return "", errors.New("Rel: can't make " + t
238     }
239     // Position base[b0:bi] and targ[t0:ti] at the first

```

```

240     bl := len(base)
241     tl := len(targ)
242     var b0, bi, t0, ti int
243     for {
244         for bi < bl && base[bi] != Separator {
245             bi++
246         }
247         for ti < tl && targ[ti] != Separator {
248             ti++
249         }
250         if targ[t0:ti] != base[b0:bi] {
251             break
252         }
253         if bi < bl {
254             bi++
255         }
256         if ti < tl {
257             ti++
258         }
259         b0 = bi
260         t0 = ti
261     }
262     if base[b0:bi] == ".." {
263         return "", errors.New("Rel: can't make " + t
264     }
265     if b0 != bl {
266         // Base elements left. Must go up before goi
267         seps := strings.Count(base[b0:bl], string(Separator))
268         size := 2 + seps*3
269         if tl != t0 {
270             size += 1 + tl - t0
271         }
272         buf := make([]byte, size)
273         n := copy(buf, "..")
274         for i := 0; i < seps; i++ {
275             buf[n] = Separator
276             copy(buf[n+1:], "..")
277             n += 3
278         }
279         if t0 != tl {
280             buf[n] = Separator
281             copy(buf[n+1:], targ[t0:])
282         }
283         return string(buf), nil
284     }
285     return targ[t0:], nil
286 }
287
288 // SkipDir is used as a return value from WalkFuncs to indic

```

```

289 // the directory named in the call is to be skipped. It is n
290 // as an error by any function.
291 var SkipDir = errors.New("skip this directory")
292
293 // WalkFunc is the type of the function called for each file
294 // visited by Walk. If there was a problem walking to the f
295 // named by path, the incoming error will describe the probl
296 // function can decide how to handle that error (and Walk wi
297 // into that directory). If an error is returned, processin
298 // sole exception is that if path is a directory and the fun
299 // special value SkipDir, the contents of the directory are
300 // and processing continues as usual on the next file.
301 type WalkFunc func(path string, info os.FileInfo, err error)
302
303 // walk recursively descends path, calling w.
304 func walk(path string, info os.FileInfo, walkFn WalkFunc) er
305     err := walkFn(path, info, nil)
306     if err != nil {
307         if info.IsDir() && err == SkipDir {
308             return nil
309         }
310         return err
311     }
312
313     if !info.IsDir() {
314         return nil
315     }
316
317     list, err := readDir(path)
318     if err != nil {
319         return walkFn(path, info, err)
320     }
321
322     for _, fileInfo := range list {
323         if err = walk(Join(path, fileInfo.Name()), f
324             return err
325         }
326     }
327     return nil
328 }
329
330 // Walk walks the file tree rooted at root, calling walkFn f
331 // directory in the tree, including root. All errors that ar
332 // and directories are filtered by walkFn. The files are wal
333 // order, which makes the output deterministic but means tha
334 // large directories Walk can be inefficient.
335 func Walk(root string, walkFn WalkFunc) error {
336     info, err := os.Lstat(root)
337     if err != nil {

```

```

338         return walkFn(root, nil, err)
339     }
340     return walk(root, info, walkFn)
341 }
342
343 // readDir reads the directory named by dirname and returns
344 // a sorted list of directory entries.
345 // Copied from io/ioutil to avoid the circular import.
346 func readDir(dirname string) ([]os.FileInfo, error) {
347     f, err := os.Open(dirname)
348     if err != nil {
349         return nil, err
350     }
351     list, err := f.Readdir(-1)
352     f.Close()
353     if err != nil {
354         return nil, err
355     }
356     sort.Sort(byName(list))
357     return list, nil
358 }
359
360 // byName implements sort.Interface.
361 type byName []os.FileInfo
362
363 func (f byName) Len() int           { return len(f) }
364 func (f byName) Less(i, j int) bool { return f[i].Name() < f[j].Name() }
365 func (f byName) Swap(i, j int)      { f[i], f[j] = f[j], f[i] }
366
367 // Base returns the last element of path.
368 // Trailing path separators are removed before extracting the last element.
369 // If the path is empty, Base returns ".".
370 // If the path consists entirely of separators, Base returns the separator.
371 func Base(path string) string {
372     if path == "" {
373         return "."
374     }
375     // Strip trailing slashes.
376     for len(path) > 0 && os.IsPathSeparator(path[len(path)-1]) {
377         path = path[:len(path)-1]
378     }
379     // Throw away volume name
380     path = path[len(VolumeName(path)):]
381     // Find the last element
382     i := len(path) - 1
383     for i >= 0 && !os.IsPathSeparator(path[i]) {
384         i--
385     }
386     if i >= 0 {
387         path = path[i+1:]

```

```

388     }
389     // If empty now, it had only slashes.
390     if path == "" {
391         return string(Separator)
392     }
393     return path
394 }
395
396 // Dir returns all but the last element of path, typically t
397 // Trailing path separators are removed before processing.
398 // If the path is empty, Dir returns ".".
399 // If the path consists entirely of separators, Dir returns
400 // The returned path does not end in a separator unless it i
401 func Dir(path string) string {
402     vol := VolumeName(path)
403     i := len(path) - 1
404     for i >= len(vol) && !os.IsPathSeparator(path[i]) {
405         i--
406     }
407     dir := Clean(path[len(vol) : i+1])
408     last := len(dir) - 1
409     if last > 0 && os.IsPathSeparator(dir[last]) {
410         dir = dir[:last]
411     }
412     if dir == "" {
413         dir = "."
414     }
415     return vol + dir
416 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/path/filepath/path_unix.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package filepath
8
9 import "strings"
10
11 // IsAbs returns true if the path is absolute.
12 func IsAbs(path string) bool {
13     return strings.HasPrefix(path, "/")
14 }
15
16 // VolumeName returns the leading volume name on Windows.
17 // It returns "" elsewhere.
18 func VolumeName(path string) string {
19     return ""
20 }
21
22 // HasPrefix exists for historical compatibility and should
23 func HasPrefix(p, prefix string) bool {
24     return strings.HasPrefix(p, prefix)
25 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/path/filepath/symlink.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build !windows
6
7 package filepath
8
9 import (
10     "bytes"
11     "errors"
12     "os"
13     "strings"
14 )
15
16 func evalSymlinks(path string) (string, error) {
17     const maxIter = 255
18     originalPath := path
19     // consume path by taking each frontmost path element
20     // expanding it if it's a symlink, and appending it
21     var b bytes.Buffer
22     for n := 0; path != ""; n++ {
23         if n > maxIter {
24             return "", errors.New("EvalSymlinks:
25         }
26
27         // find next path component, p
28         i := strings.IndexRune(path, Separator)
29         var p string
30         if i == -1 {
31             p, path = path, ""
32         } else {
33             p, path = path[:i], path[i+1:]
34         }
35
36         if p == "" {
37             if b.Len() == 0 {
38                 // must be absolute path
39                 b.WriteRune(Separator)
40             }
41             continue

```

```

42     }
43
44     fi, err := os.Lstat(b.String() + p)
45     if err != nil {
46         return "", err
47     }
48     if fi.Mode()&os.ModeSymlink == 0 {
49         b.WriteString(p)
50         if path != "" {
51             b.WriteRune(Separator)
52         }
53         continue
54     }
55
56     // it's a symlink, put it at the front of pa
57     dest, err := os.Readlink(b.String() + p)
58     if err != nil {
59         return "", err
60     }
61     if IsAbs(dest) {
62         b.Reset()
63     }
64     path = dest + string(Separator) + path
65 }
66 return Clean(b.String()), nil
67 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/reflect/deepequal.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Deep equality test via reflection
6
7 package reflect
8
9 // During deepValueEqual, must keep track of checks that are
10 // in progress. The comparison algorithm assumes that all
11 // checks in progress are true when it reencounters them.
12 // Visited are stored in a map indexed by 17 * a1 + a2;
13 type visit struct {
14     a1    uintptr
15     a2    uintptr
16     typ  Type
17     next *visit
18 }
19
20 // Tests for deep equality using reflected types. The map ar
21 // comparisons that have already been seen, which allows sho
22 // recursive types.
23 func deepValueEqual(v1, v2 Value, visited map[uintptr]*visit
24     if !v1.IsValid() || !v2.IsValid() {
25         return v1.IsValid() == v2.IsValid()
26     }
27     if v1.Type() != v2.Type() {
28         return false
29     }
30
31     // if depth > 10 { panic("deepValueEqual") } // f
32
33     if v1.CanAddr() && v2.CanAddr() {
34         addr1 := v1.UnsafeAddr()
35         addr2 := v2.UnsafeAddr()
36         if addr1 > addr2 {
37             // Canonicalize order to reduce numb
38             addr1, addr2 = addr2, addr1
39         }
40
41         // Short circuit if references are identical
```

```

42         if addr1 == addr2 {
43             return true
44         }
45
46         // ... or already seen
47         h := 17*addr1 + addr2
48         seen := visited[h]
49         typ := v1.Type()
50         for p := seen; p != nil; p = p.next {
51             if p.a1 == addr1 && p.a2 == addr2 &&
52                 return true
53         }
54     }
55
56     // Remember for later.
57     visited[h] = &visit{addr1, addr2, typ, seen}
58 }
59
60 switch v1.Kind() {
61 case Array:
62     if v1.Len() != v2.Len() {
63         return false
64     }
65     for i := 0; i < v1.Len(); i++ {
66         if !deepValueEqual(v1.Index(i), v2.I
67             return false
68         }
69     }
70     return true
71 case Slice:
72     if v1.IsNil() != v2.IsNil() {
73         return false
74     }
75     if v1.Len() != v2.Len() {
76         return false
77     }
78     for i := 0; i < v1.Len(); i++ {
79         if !deepValueEqual(v1.Index(i), v2.I
80             return false
81         }
82     }
83     return true
84 case Interface:
85     if v1.IsNil() || v2.IsNil() {
86         return v1.IsNil() == v2.IsNil()
87     }
88     return deepValueEqual(v1.Elem(), v2.Elem()),
89 case Ptr:
90     return deepValueEqual(v1.Elem(), v2.Elem()),
91 case Struct:

```

```

92         for i, n := 0, v1.NumField(); i < n; i++ {
93             if !deepValueEqual(v1.Field(i), v2.F
94                 return false
95             }
96         }
97         return true
98     case Map:
99         if v1.IsNil() != v2.IsNil() {
100             return false
101         }
102         if v1.Len() != v2.Len() {
103             return false
104         }
105         for _, k := range v1.MapKeys() {
106             if !deepValueEqual(v1.MapIndex(k), v
107                 return false
108             }
109         }
110         return true
111     case Func:
112         if v1.IsNil() && v2.IsNil() {
113             return true
114         }
115         // Can't do better than this:
116         return false
117     default:
118         // Normal equality suffices
119         return valueInterface(v1, false) == valueInt
120     }
121     panic("Not reached")
122 }
123 }
124
125 // DeepEqual tests for deep equality. It uses normal == equa
126 // but will scan members of arrays, slices, maps, and fields
127 // handles recursive types. Functions are equal only if they
128 func DeepEqual(a1, a2 interface{}) bool {
129     if a1 == nil || a2 == nil {
130         return a1 == a2
131     }
132     v1 := ValueOf(a1)
133     v2 := ValueOf(a2)
134     if v1.Type() != v2.Type() {
135         return false
136     }
137     return deepValueEqual(v1, v2, make(map[uintptr]*visi
138 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/reflect/type.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package reflect implements run-time reflection, allowing
6 // manipulate objects with arbitrary types. The typical use
7 // with static type interface{} and extract its dynamic type
8 // calling TypeOf, which returns a Type.
9 //
10 // A call to ValueOf returns a Value representing the run-ti
11 // Zero takes a Type and returns a Value representing a zero
12 // for that type.
13 //
14 // See "The Laws of Reflection" for an introduction to refle
15 // http://golang.org/doc/articles/laws\_of\_reflection.html
16 package reflect
17
18 import (
19     "strconv"
20     "sync"
21     "unsafe"
22 )
23
24 // Type is the representation of a Go type.
25 //
26 // Not all methods apply to all kinds of types. Restriction
27 // if any, are noted in the documentation for each method.
28 // Use the Kind method to find out the kind of type before
29 // calling kind-specific methods. Calling a method
30 // inappropriate to the kind of type causes a run-time panic
31 type Type interface {
32     // Methods applicable to all types.
33
34     // Align returns the alignment in bytes of a value o
35     // this type when allocated in memory.
36     Align() int
37
38     // FieldAlign returns the alignment in bytes of a va
39     // this type when used as a field in a struct.
40     FieldAlign() int
41
42     // Method returns the i'th method in the type's meth
43     // It panics if i is not in the range [0, NumMethod(
44     //
```

```
45 // For a non-interface type T or *T, the returned Me
46 // fields describe a function whose first argument i
47 //
48 // For an interface type, the returned Method's Type
49 // method signature, without a receiver, and the Fun
50 Method(int) Method
51
52 // MethodByName returns the method with that name in
53 // method set and a boolean indicating if the method
54 //
55 // For a non-interface type T or *T, the returned Me
56 // fields describe a function whose first argument i
57 //
58 // For an interface type, the returned Method's Type
59 // method signature, without a receiver, and the Fun
60 MethodByName(string) (Method, bool)
61
62 // NumMethod returns the number of methods in the ty
63 NumMethod() int
64
65 // Name returns the type's name within its package.
66 // It returns an empty string for unnamed types.
67 Name() string
68
69 // PkgPath returns a named type's package path, that
70 // that uniquely identifies the package, such as "en
71 // If the type was predeclared (string, error) or un
72 // the package path will be the empty string.
73 PkgPath() string
74
75 // Size returns the number of bytes needed to store
76 // a value of the given type; it is analogous to uns
77 Size() uintptr
78
79 // String returns a string representation of the typ
80 // The string representation may use shortened packa
81 // (e.g., base64 instead of "encoding/base64") and i
82 // guaranteed to be unique among types. To test for
83 // compare the Types directly.
84 String() string
85
86 // Kind returns the specific kind of this type.
87 Kind() Kind
88
89 // Implements returns true if the type implements th
90 Implements(u Type) bool
91
92 // AssignableTo returns true if a value of the type
93 AssignableTo(u Type) bool
94
```

```

95 // Methods applicable only to some types, depending
96 // The methods allowed for each kind are:
97 //
98 //     Int*, Uint*, Float*, Complex*: Bits
99 //     Array: Elem, Len
100 //     Chan: ChanDir, Elem
101 //     Func: In, NumIn, Out, NumOut, IsVariadic.
102 //     Map: Key, Elem
103 //     Ptr: Elem
104 //     Slice: Elem
105 //     Struct: Field, FieldByIndex, FieldByName, Fi
106
107 // Bits returns the size of the type in bits.
108 // It panics if the type's Kind is not one of the
109 // sized or unsized Int, Uint, Float, or Complex kin
110 Bits() int
111
112 // ChanDir returns a channel type's direction.
113 // It panics if the type's Kind is not Chan.
114 ChanDir() ChanDir
115
116 // IsVariadic returns true if a function type's fina
117 // is a "..." parameter. If so, t.In(t.NumIn() - 1)
118 // implicit actual type []T.
119 //
120 // For concreteness, if t represents func(x int, y .
121 //
122 //     t.NumIn() == 2
123 //     t.In(0) is the reflect.Type for "int"
124 //     t.In(1) is the reflect.Type for "[]float64"
125 //     t.IsVariadic() == true
126 //
127 // IsVariadic panics if the type's Kind is not Func.
128 IsVariadic() bool
129
130 // Elem returns a type's element type.
131 // It panics if the type's Kind is not Array, Chan,
132 Elem() Type
133
134 // Field returns a struct type's i'th field.
135 // It panics if the type's Kind is not Struct.
136 // It panics if i is not in the range [0, NumField())
137 Field(i int) StructField
138
139 // FieldByIndex returns the nested field correspondi
140 // to the index sequence. It is equivalent to calli
141 // successively for each index i.
142 // It panics if the type's Kind is not Struct.
143 FieldByIndex(index []int) StructField

```

```

144
145 // FieldByName returns the struct field with the giv
146 // and a boolean indicating if the field was found.
147 FieldByName(name string) (StructField, bool)
148
149 // FieldByNameFunc returns the first struct field wi
150 // that satisfies the match function and a boolean i
151 // the field was found.
152 FieldByNameFunc(match func(string) bool) (StructFiel
153
154 // In returns the type of a function type's i'th inp
155 // It panics if the type's Kind is not Func.
156 // It panics if i is not in the range [0, NumIn()).
157 In(i int) Type
158
159 // Key returns a map type's key type.
160 // It panics if the type's Kind is not Map.
161 Key() Type
162
163 // Len returns an array type's length.
164 // It panics if the type's Kind is not Array.
165 Len() int
166
167 // NumField returns a struct type's field count.
168 // It panics if the type's Kind is not Struct.
169 NumField() int
170
171 // NumIn returns a function type's input parameter c
172 // It panics if the type's Kind is not Func.
173 NumIn() int
174
175 // NumOut returns a function type's output parameter
176 // It panics if the type's Kind is not Func.
177 NumOut() int
178
179 // Out returns the type of a function type's i'th ou
180 // It panics if the type's Kind is not Func.
181 // It panics if i is not in the range [0, NumOut()).
182 Out(i int) Type
183
184 runtimeType() *runtimeType
185 common() *commonType
186 uncommon() *uncommonType
187 }
188
189 // A Kind represents the specific kind of type that a Type r
190 // The zero Kind is not a valid kind.
191 type Kind uint
192

```

```

193 const (
194     Invalid Kind = iota
195     Bool
196     Int
197     Int8
198     Int16
199     Int32
200     Int64
201     Uint
202     Uint8
203     Uint16
204     Uint32
205     Uint64
206     Uintptr
207     Float32
208     Float64
209     Complex64
210     Complex128
211     Array
212     Chan
213     Func
214     Interface
215     Map
216     Ptr
217     Slice
218     String
219     Struct
220     UnsafePointer
221 )
222
223 /*
224  * These data structures are known to the compiler (../../cm
225  * A few are known to ../runtime/type.go to convey to debugg
226  */
227
228 // The compiler can only construct empty interface values at
229 // compile time; non-empty interface values get created
230 // during initialization. Type is an empty interface
231 // so that the compiler can lay out references as data.
232 // The underlying type is *reflect.ArrayType and so on.
233 type runtimeType interface{}
234
235 // commonType is the common implementation of most values.
236 // It is embedded in other, public struct types, but always
237 // with a unique tag like `reflect:"array"` or `reflect:"ptr
238 // so that code cannot convert from, say, *arrayType to *ptr
239 type commonType struct {
240     size      uintptr    // size in bytes
241     hash      uint32     // hash of type; avoids c
242     _         uint8     // unused/padding

```

```

243         align          uint8          // alignment of variable
244         fieldAlign     uint8          // alignment of struct fi
245         kind           uint8          // enumeration for C
246         alg            *uintptr       // algorithm table (../ru
247         string         *string        // string form; unnecessa
248         *uncommonType // (relatively) uncommon
249         ptrToThis      *runtimeType   // pointer to this type,
250     }
251
252 // Method on non-interface type
253 type method struct {
254     name      *string      // name of method
255     pkgPath   *string      // nil for exported Names; ot
256     mtyp      *runtimeType // method type (without recei
257     typ       *runtimeType //.(*FuncType) underneath (w
258     ifn       unsafe.Pointer // fn used in interface call
259     tfn       unsafe.Pointer // fn used for normal method
260 }
261
262 // uncommonType is present only for types with names or meth
263 // (if T is a named type, the uncommonTypes for T and *T hav
264 // Using a pointer to this struct reduces the overall size r
265 // to describe an unnamed type with no methods.
266 type uncommonType struct {
267     name      *string // name of type
268     pkgPath   *string // import path; nil for built-in ty
269     methods  []method // methods associated with type
270 }
271
272 // ChanDir represents a channel type's direction.
273 type ChanDir int
274
275 const (
276     RecvDir ChanDir          = 1 << iota // <-chan
277     SendDir                               // chan<-
278     BothDir = RecvDir | SendDir          // chan
279 )
280
281 // arrayType represents a fixed array type.
282 type arrayType struct {
283     commonType `reflect:"array"`
284     elem       *runtimeType // array element type
285     slice      *runtimeType // slice type
286     len        uintptr
287 }
288
289 // chanType represents a channel type.
290 type chanType struct {
291     commonType `reflect:"chan"`

```

```

292         elem      *runtimeType // channel element type
293         dir        uintptr      // channel direction (ChanDi
294     }
295
296 // funcType represents a function type.
297 type funcType struct {
298     commonType `reflect:"func"`
299     dotdotdot  bool             // last input parameter is
300     in         []*runtimeType // input parameter types
301     out        []*runtimeType // output parameter types
302 }
303
304 // imethod represents a method on an interface type
305 type imethod struct {
306     name      *string // name of method
307     pkgPath  *string // nil for exported Names; othe
308     typ      *runtimeType // .(*FuncType) underneath
309 }
310
311 // interfaceType represents an interface type.
312 type interfaceType struct {
313     commonType `reflect:"interface"`
314     methods    []imethod // sorted by hash
315 }
316
317 // mapType represents a map type.
318 type mapType struct {
319     commonType `reflect:"map"`
320     key        *runtimeType // map key type
321     elem       *runtimeType // map element (value) type
322 }
323
324 // ptrType represents a pointer type.
325 type ptrType struct {
326     commonType `reflect:"ptr"`
327     elem       *runtimeType // pointer element (pointed
328 }
329
330 // sliceType represents a slice type.
331 type sliceType struct {
332     commonType `reflect:"slice"`
333     elem       *runtimeType // slice element type
334 }
335
336 // Struct field
337 type structField struct {
338     name      *string // nil for embedded fields
339     pkgPath  *string // nil for exported Names; othe
340     typ      *runtimeType // type of field

```

```

341         tag      *string      // nil if no tag
342         offset  uintptr      // byte offset of field within
343     }
344
345 // structType represents a struct type.
346 type structType struct {
347     commonType `reflect:"struct"`
348     fields     []structField // sorted by offset
349 }
350
351 /*
352  * The compiler knows the exact layout of all the data struc
353  * The compiler does not know about the data structures and
354  */
355
356 // Method represents a single method.
357 type Method struct {
358     // Name is the method name.
359     // PkgPath is the package path that qualifies a lowe
360     // method name. It is empty for upper case (exporte
361     // The combination of PkgPath and Name uniquely iden
362     // in a method set.
363     // See http://golang.org/ref/spec#Uniqueness\_of\_iden
364     Name      string
365     PkgPath   string
366
367     Type     Type // method type
368     Func     Value // func with receiver as first argument
369     Index    int  // index for Type.Method
370 }
371
372 // High bit says whether type has
373 // embedded pointers, to help garbage collector.
374 const kindMask = 0x7f
375
376 func (k Kind) String() string {
377     if int(k) < len(kindNames) {
378         return kindNames[k]
379     }
380     return "kind" + strconv.Itoa(int(k))
381 }
382
383 var kindNames = []string{
384     Invalid:    "invalid",
385     Bool:      "bool",
386     Int:       "int",
387     Int8:     "int8",
388     Int16:    "int16",
389     Int32:    "int32",
390     Int64:    "int64",

```

```

391         Uint:           "uint",
392         Uint8:          "uint8",
393         Uint16:         "uint16",
394         Uint32:         "uint32",
395         Uint64:         "uint64",
396         Uintptr:        "uintptr",
397         Float32:        "float32",
398         Float64:        "float64",
399         Complex64:      "complex64",
400         Complex128:    "complex128",
401         Array:          "array",
402         Chan:           "chan",
403         Func:           "func",
404         Interface:     "interface",
405         Map:            "map",
406         Ptr:            "ptr",
407         Slice:          "slice",
408         String:         "string",
409         Struct:         "struct",
410         UnsafePointer: "unsafe.Pointer",
411     }
412
413     func (t *uncommonType) uncommon() *uncommonType {
414         return t
415     }
416
417     func (t *uncommonType) PkgPath() string {
418         if t == nil || t.pkgPath == nil {
419             return ""
420         }
421         return *t.pkgPath
422     }
423
424     func (t *uncommonType) Name() string {
425         if t == nil || t.name == nil {
426             return ""
427         }
428         return *t.name
429     }
430
431     func (t *commonType) toType() Type {
432         if t == nil {
433             return nil
434         }
435         return t
436     }
437
438     func (t *commonType) String() string { return *t.string }
439

```

```

440 func (t *commonType) Size() uintptr { return t.size }
441
442 func (t *commonType) Bits() int {
443     if t == nil {
444         panic("reflect: Bits of nil Type")
445     }
446     k := t.Kind()
447     if k < Int || k > Complex128 {
448         panic("reflect: Bits of non-arithmetic Type")
449     }
450     return int(t.size) * 8
451 }
452
453 func (t *commonType) Align() int { return int(t.align) }
454
455 func (t *commonType) FieldAlign() int { return int(t.fieldAl
456
457 func (t *commonType) Kind() Kind { return Kind(t.kind & kind
458
459 func (t *commonType) common() *commonType { return t }
460
461 func (t *uncommonType) Method(i int) (m Method) {
462     if t == nil || i < 0 || i >= len(t.methods) {
463         panic("reflect: Method index out of range")
464     }
465     p := &t.methods[i]
466     if p.name != nil {
467         m.Name = *p.name
468     }
469     fl := flag(Func) << flagKindShift
470     if p.pkgPath != nil {
471         m.PkgPath = *p.pkgPath
472         fl |= flagR0
473     }
474     mt := toCommonType(p.typ)
475     m.Type = mt
476     fn := p.tfn
477     m.Func = Value{mt, fn, fl}
478     m.Index = i
479     return
480 }
481
482 func (t *uncommonType) NumMethod() int {
483     if t == nil {
484         return 0
485     }
486     return len(t.methods)
487 }
488

```

```

489 func (t *uncommonType) MethodByName(name string) (m Method,
490     if t == nil {
491         return
492     }
493     var p *method
494     for i := range t.methods {
495         p = &t.methods[i]
496         if p.name != nil && *p.name == name {
497             return t.Method(i), true
498         }
499     }
500     return
501 }
502
503 // TODO(rsc): 6g supplies these, but they are not
504 // as efficient as they could be: they have commonType
505 // as the receiver instead of *commonType.
506 func (t *commonType) NumMethod() int {
507     if t.Kind() == Interface {
508         tt := (*interfaceType)(unsafe.Pointer(t))
509         return tt.NumMethod()
510     }
511     return t.uncommonType.NumMethod()
512 }
513
514 func (t *commonType) Method(i int) (m Method) {
515     if t.Kind() == Interface {
516         tt := (*interfaceType)(unsafe.Pointer(t))
517         return tt.Method(i)
518     }
519     return t.uncommonType.Method(i)
520 }
521
522 func (t *commonType) MethodByName(name string) (m Method, ok
523     if t.Kind() == Interface {
524         tt := (*interfaceType)(unsafe.Pointer(t))
525         return tt.MethodByName(name)
526     }
527     return t.uncommonType.MethodByName(name)
528 }
529
530 func (t *commonType) PkgPath() string {
531     return t.uncommonType.PkgPath()
532 }
533
534 func (t *commonType) Name() string {
535     return t.uncommonType.Name()
536 }
537
538 func (t *commonType) ChanDir() ChanDir {

```

```

539         if t.Kind() != Chan {
540             panic("reflect: ChanDir of non-chan type")
541         }
542         tt := (*chanType)(unsafe.Pointer(t))
543         return ChanDir(tt.dir)
544     }
545
546     func (t *commonType) IsVariadic() bool {
547         if t.Kind() != Func {
548             panic("reflect: IsVariadic of non-func type")
549         }
550         tt := (*funcType)(unsafe.Pointer(t))
551         return tt.dotdotdot
552     }
553
554     func (t *commonType) Elem() Type {
555         switch t.Kind() {
556         case Array:
557             tt := (*arrayType)(unsafe.Pointer(t))
558             return toType(tt.elem)
559         case Chan:
560             tt := (*chanType)(unsafe.Pointer(t))
561             return toType(tt.elem)
562         case Map:
563             tt := (*mapType)(unsafe.Pointer(t))
564             return toType(tt.elem)
565         case Ptr:
566             tt := (*ptrType)(unsafe.Pointer(t))
567             return toType(tt.elem)
568         case Slice:
569             tt := (*sliceType)(unsafe.Pointer(t))
570             return toType(tt.elem)
571         }
572         panic("reflect: Elem of invalid type")
573     }
574
575     func (t *commonType) Field(i int) StructField {
576         if t.Kind() != Struct {
577             panic("reflect: Field of non-struct type")
578         }
579         tt := (*structType)(unsafe.Pointer(t))
580         return tt.Field(i)
581     }
582
583     func (t *commonType) FieldByIndex(index []int) StructField {
584         if t.Kind() != Struct {
585             panic("reflect: FieldByIndex of non-struct t
586         }
587         tt := (*structType)(unsafe.Pointer(t))

```

```

588         return tt.FieldByIndex(index)
589     }
590
591     func (t *commonType) FieldByName(name string) (StructField,
592         if t.Kind() != Struct {
593             panic("reflect: FieldByName of non-struct ty
594         }
595         tt := (*structType)(unsafe.Pointer(t))
596         return tt.FieldByName(name)
597     }
598
599     func (t *commonType) FieldByNameFunc(match func(string) bool
600         if t.Kind() != Struct {
601             panic("reflect: FieldByNameFunc of non-struct
602         }
603         tt := (*structType)(unsafe.Pointer(t))
604         return tt.FieldByNameFunc(match)
605     }
606
607     func (t *commonType) In(i int) Type {
608         if t.Kind() != Func {
609             panic("reflect: In of non-func type")
610         }
611         tt := (*funcType)(unsafe.Pointer(t))
612         return toType(tt.in[i])
613     }
614
615     func (t *commonType) Key() Type {
616         if t.Kind() != Map {
617             panic("reflect: Key of non-map type")
618         }
619         tt := (*mapType)(unsafe.Pointer(t))
620         return toType(tt.key)
621     }
622
623     func (t *commonType) Len() int {
624         if t.Kind() != Array {
625             panic("reflect: Len of non-array type")
626         }
627         tt := (*arrayType)(unsafe.Pointer(t))
628         return int(tt.len)
629     }
630
631     func (t *commonType) NumField() int {
632         if t.Kind() != Struct {
633             panic("reflect: NumField of non-struct type")
634         }
635         tt := (*structType)(unsafe.Pointer(t))
636         return len(tt.fields)

```

```

637 }
638
639 func (t *commonType) NumIn() int {
640     if t.Kind() != Func {
641         panic("reflect: NumIn of non-func type")
642     }
643     tt := (*funcType)(unsafe.Pointer(t))
644     return len(tt.in)
645 }
646
647 func (t *commonType) NumOut() int {
648     if t.Kind() != Func {
649         panic("reflect: NumOut of non-func type")
650     }
651     tt := (*funcType)(unsafe.Pointer(t))
652     return len(tt.out)
653 }
654
655 func (t *commonType) Out(i int) Type {
656     if t.Kind() != Func {
657         panic("reflect: Out of non-func type")
658     }
659     tt := (*funcType)(unsafe.Pointer(t))
660     return toType(tt.out[i])
661 }
662
663 func (d ChanDir) String() string {
664     switch d {
665     case SendDir:
666         return "chan<-"
667     case RecvDir:
668         return "<-chan"
669     case BothDir:
670         return "chan"
671     }
672     return "ChanDir" + strconv.Itoa(int(d))
673 }
674
675 // Method returns the i'th method in the type's method set.
676 func (t *interfaceType) Method(i int) (m Method) {
677     if i < 0 || i >= len(t.methods) {
678         return
679     }
680     p := &t.methods[i]
681     m.Name = *p.name
682     if p.pkgPath != nil {
683         m.PkgPath = *p.pkgPath
684     }
685     m.Type = toType(p.typ)
686     m.Index = i

```

```

687         return
688     }
689
690 // NumMethod returns the number of interface methods in the
691 func (t *interfaceType) NumMethod() int { return len(t.metho
692
693 // MethodByName method with the given name in the type's met
694 func (t *interfaceType) MethodByName(name string) (m Method,
695     if t == nil {
696         return
697     }
698     var p *imethod
699     for i := range t.methods {
700         p = &t.methods[i]
701         if *p.name == name {
702             return t.Method(i), true
703         }
704     }
705     return
706 }
707
708 // A StructField describes a single field in a struct.
709 type StructField struct {
710     // Name is the field name.
711     // PkgPath is the package path that qualifies a lowe
712     // field name. It is empty for upper case (exported
713     // See http://golang.org/ref/spec#Uniqueness\_of\_iden
714     Name     string
715     PkgPath  string
716
717     Type     Type     // field type
718     Tag     StructTag // field tag string
719     Offset  uintptr  // offset within struct, in byte
720     Index   []int   // index sequence for Type.Field
721     Anonymous bool     // is an anonymous field
722 }
723
724 // A StructTag is the tag string in a struct field.
725 //
726 // By convention, tag strings are a concatenation of
727 // optionally space-separated key:"value" pairs.
728 // Each key is a non-empty string consisting of non-control
729 // characters other than space (U+0020 ' '), quote (U+0022 '
730 // and colon (U+003A ':'). Each value is quoted using U+002
731 // characters and Go string literal syntax.
732 type StructTag string
733
734 // Get returns the value associated with key in the tag stri
735 // If there is no such key in the tag, Get returns the empty

```

```

736 // If the tag does not have the conventional format, the val
737 // returned by Get is unspecified.
738 func (tag StructTag) Get(key string) string {
739     for tag != "" {
740         // skip leading space
741         i := 0
742         for i < len(tag) && tag[i] == ' ' {
743             i++
744         }
745         tag = tag[i:]
746         if tag == "" {
747             break
748         }
749
750         // scan to colon.
751         // a space or a quote is a syntax error
752         i = 0
753         for i < len(tag) && tag[i] != ' ' && tag[i]
754             i++
755         }
756         if i+1 >= len(tag) || tag[i] != ':' || tag[i]
757             break
758         }
759         name := string(tag[:i])
760         tag = tag[i+1:]
761
762         // scan quoted string to find value
763         i = 1
764         for i < len(tag) && tag[i] != '"' {
765             if tag[i] == '\\' {
766                 i++
767             }
768             i++
769         }
770         if i >= len(tag) {
771             break
772         }
773         qvalue := string(tag[:i+1])
774         tag = tag[i+1:]
775
776         if key == name {
777             value, _ := strconv.Unquote(qvalue)
778             return value
779         }
780     }
781     return ""
782 }
783
784 // Field returns the i'th struct field.

```

```

785 func (t *structType) Field(i int) (f StructField) {
786     if i < 0 || i >= len(t.fields) {
787         return
788     }
789     p := &t.fields[i]
790     f.Type = toType(p.typ)
791     if p.name != nil {
792         f.Name = *p.name
793     } else {
794         t := f.Type
795         if t.Kind() == Ptr {
796             t = t.Elem()
797         }
798         f.Name = t.Name()
799         f.Anonymous = true
800     }
801     if p.pkgPath != nil {
802         f.PkgPath = *p.pkgPath
803     }
804     if p.tag != nil {
805         f.Tag = StructTag(*p.tag)
806     }
807     f.Offset = p.offset
808
809     // NOTE(rsc): This is the only allocation in the int
810     // presented by a reflect.Type. It would be nice to
811     // at least in the common cases, but we need to make
812     // that misbehaving clients of reflect cannot affect
813     // uses of reflect. One possibility is CL 5371098,
814     // postponed that ugliness until there is a demonstr
815     // need for the performance. This is issue 2320.
816     f.Index = []int{i}
817     return
818 }
819
820 // TODO(gri): Should there be an error/bool indicator if the
821 // is wrong for FieldByIndex?
822
823 // FieldByIndex returns the nested field corresponding to in
824 func (t *structType) FieldByIndex(index []int) (f StructFiel
825     f.Type = Type(t.toType())
826     for i, x := range index {
827         if i > 0 {
828             ft := f.Type
829             if ft.Kind() == Ptr && ft.Elem().Kin
830                 ft = ft.Elem()
831             }
832             f.Type = ft
833         }
834     f = f.Type.Field(x)

```

```

835     }
836     return
837 }
838
839 const inf = 1 << 30 // infinity - no struct has that many ne
840
841 func (t *structType) fieldByNameFunc(match func(string) bool
842     fd = inf // field depth
843
844     if mark[t] {
845         // Struct already seen.
846         return
847     }
848     mark[t] = true
849
850     var fi int // field index
851     n := 0     // number of matching fields at depth fd
852 L:
853     for i := range t.fields {
854         f := t.Field(i)
855         d := inf
856         switch {
857         case match(f.Name):
858             // Matching top-level field.
859             d = depth
860         case f.Anonymous:
861             ft := f.Type
862             if ft.Kind() == Ptr {
863                 ft = ft.Elem()
864             }
865             switch {
866             case match(ft.Name()):
867                 // Matching anonymous top-le
868                 d = depth
869             case fd > depth:
870                 // No top-level field yet; l
871                 if ft.Kind() == Struct {
872                     st := (*structType)(
873                         f, d = st.fieldByNar
874                 }
875             }
876         }
877
878         switch {
879         case d < fd:
880             // Found field at shallower depth.
881             ff, fi, fd = f, i, d
882             n = 1
883         case d == fd:

```

```

884             // More than one matching field at t
885             // Same as no field found at this de
886             n++
887             if d == depth {
888                 // Impossible to find a fiel
889                 break L
890             }
891         }
892     }
893
894     if n == 1 {
895         // Found matching field.
896         if depth >= len(ff.Index) {
897             ff.Index = make([]int, depth+1)
898         }
899         if len(ff.Index) > 1 {
900             ff.Index[depth] = fi
901         }
902     } else {
903         // None or more than one matching field foun
904         fd = inf
905     }
906
907     delete(mark, t)
908     return
909 }
910
911 // FieldByName returns the struct field with the given name
912 // and a boolean to indicate if the field was found.
913 func (t *structType) FieldByName(name string) (f StructField
914     return t.FieldByNameFunc(func(s string) bool { retur
915 }
916
917 // FieldByNameFunc returns the struct field with a name that
918 // match function and a boolean to indicate if the field was
919 func (t *structType) FieldByNameFunc(match func(string) bool
920     if ff, fd := t.fieldByNameFunc(match, make(map[*stru
921         ff.Index = ff.Index[0 : fd+1]
922         f, present = ff, true
923     }
924     return
925 }
926
927 // Convert runtime type to reflect type.
928 func toCommonType(p *runtimeType) *commonType {
929     if p == nil {
930         return nil
931     }
932     return (*p).(*commonType)

```

```

933 }
934
935 func toType(p *runtimeType) Type {
936     if p == nil {
937         return nil
938     }
939     return (*p).(*commonType)
940 }
941
942 // TypeOf returns the reflection Type of the value in the in
943 // TypeOf(nil) returns nil.
944 func TypeOf(i interface{}) Type {
945     eface := *(*emptyInterface)(unsafe.Pointer(&i))
946     return toType(eface.typ)
947 }
948
949 // ptrMap is the cache for PtrTo.
950 var ptrMap struct {
951     sync.RWMutex
952     m map[*commonType]*ptrType
953 }
954
955 func (t *commonType) runtimeType() *runtimeType {
956     // The runtimeType always precedes the commonType in
957     // Adjust pointer to find it.
958     var rt struct {
959         i runtimeType
960         ct commonType
961     }
962     return (*runtimeType)(unsafe.Pointer(uintptr(unsafe.
963 }
964
965 // PtrTo returns the pointer type with element t.
966 // For example, if t represents type Foo, PtrTo(t) represent
967 func PtrTo(t Type) Type {
968     return t.(*commonType).ptrTo()
969 }
970
971 func (ct *commonType) ptrTo() *commonType {
972     if p := ct.ptrToThis; p != nil {
973         return toCommonType(p)
974     }
975
976     // Otherwise, synthesize one.
977     // This only happens for pointers with no methods.
978     // We keep the mapping in a map on the side, because
979     // this operation is rare and a separate map lets us
980     // the type structures in read-only memory.
981     ptrMap.RLock()
982     if m := ptrMap.m; m != nil {

```

```

983         if p := m[ct]; p != nil {
984             ptrMap.RUnlock()
985             return &p.commonType
986         }
987     }
988     ptrMap.RUnlock()
989     ptrMap.Lock()
990     if ptrMap.m == nil {
991         ptrMap.m = make(map[*commonType]*ptrType)
992     }
993     p := ptrMap.m[ct]
994     if p != nil {
995         // some other goroutine won the race and cre
996         ptrMap.Unlock()
997         return &p.commonType
998     }
999
1000     var rt struct {
1001         i runtimeType
1002         ptrType
1003     }
1004     rt.i = &rt.commonType
1005
1006     // initialize p using *byte's ptrType as a prototype
1007     p = &rt.ptrType
1008     var ibyte interface{} = (*byte)(nil)
1009     bp := (*ptrType)(unsafe.Pointer((**(**runtimeType)(u
1010     *p = *bp
1011
1012     s := "*" + *ct.string
1013     p.string = &s
1014
1015     // For the type structures linked into the binary, t
1016     // compiler provides a good hash of the string.
1017     // Create a good hash for the new string by using
1018     // the FNV-1 hash's mixing function to combine the
1019     // old hash and the new "*".
1020     p.hash = ct.hash*16777619 ^ '*'
1021
1022     p.uncommonType = nil
1023     p.ptrToThis = nil
1024     p.elem = (*runtimeType)(unsafe.Pointer(uintptr(unsaf
1025
1026     ptrMap.m[ct] = p
1027     ptrMap.Unlock()
1028     return &p.commonType
1029 }
1030
1031 func (t *commonType) Implements(u Type) bool {

```

```

1032     if u == nil {
1033         panic("reflect: nil type passed to Type.Impl
1034     }
1035     if u.Kind() != Interface {
1036         panic("reflect: non-interface type passed to
1037     }
1038     return implements(u.(*commonType), t)
1039 }
1040
1041 func (t *commonType) AssignableTo(u Type) bool {
1042     if u == nil {
1043         panic("reflect: nil type passed to Type.Assi
1044     }
1045     uu := u.(*commonType)
1046     return directlyAssignable(uu, t) || implements(uu, t
1047 }
1048
1049 // implements returns true if the type V implements the inte
1050 func implements(T, V *commonType) bool {
1051     if T.Kind() != Interface {
1052         return false
1053     }
1054     t := (*interfaceType)(unsafe.Pointer(T))
1055     if len(t.methods) == 0 {
1056         return true
1057     }
1058
1059     // The same algorithm applies in both cases, but the
1060     // method tables for an interface type and a concret
1061     // are different, so the code is duplicated.
1062     // In both cases the algorithm is a linear scan over
1063     // lists - T's methods and V's methods - simultaneou
1064     // Since method tables are stored in a unique sorted
1065     // (alphabetical, with no duplicate method names), t
1066     // through V's methods must hit a match for each of
1067     // methods along the way, or else V does not impleme
1068     // This lets us run the scan in overall linear time
1069     // the quadratic time a naive search would require.
1070     // See also ../runtime/iface.c.
1071     if V.Kind() == Interface {
1072         v := (*interfaceType)(unsafe.Pointer(V))
1073         i := 0
1074         for j := 0; j < len(v.methods); j++ {
1075             tm := &t.methods[i]
1076             vm := &v.methods[j]
1077             if vm.name == tm.name && vm.pkgPath
1078                 if i++; i >= len(t.methods)
1079                 return true
1080         }

```

```

1081         }
1082     }
1083     return false
1084 }
1085
1086 v := V.uncommon()
1087 if v == nil {
1088     return false
1089 }
1090 i := 0
1091 for j := 0; j < len(v.methods); j++ {
1092     tm := &t.methods[i]
1093     vm := &v.methods[j]
1094     if vm.name == tm.name && vm.pkgPath == tm.pk
1095         if i++; i >= len(t.methods) {
1096             return true
1097         }
1098     }
1099 }
1100 return false
1101 }
1102
1103 // directlyAssignable returns true if a value x of type V ca
1104 // assigned (using memmove) to a value of type T.
1105 // http://golang.org/doc/go_spec.html#Assignability
1106 // Ignoring the interface rules (implemented elsewhere)
1107 // and the ideal constant rules (no ideal constants at run t
1108 func directlyAssignable(T, V *commonType) bool {
1109     // x's type V is identical to T?
1110     if T == V {
1111         return true
1112     }
1113
1114     // Otherwise at least one of T and V must be unnamed
1115     // and they must have the same kind.
1116     if T.Name() != "" && V.Name() != "" || T.Kind() != V
1117         return false
1118 }
1119
1120 // x's type T and V have identical underlying types.
1121 // Since at least one is unnamed, only the composite
1122 // need to be considered.
1123 switch T.Kind() {
1124 case Array:
1125     return T.Elem() == V.Elem() && T.Len() == V.
1126
1127 case Chan:
1128     // Special case:
1129     // x is a bidirectional channel value, T is
1130     // and x's type V and T have identical eleme

```

```

1131         if V.ChanDir() == BothDir && T.Elem() == V.E
1132             return true
1133     }
1134
1135     // Otherwise continue test for identical und
1136     return V.ChanDir() == T.ChanDir() && T.Elem(
1137
1138     case Func:
1139         t := (*funcType)(unsafe.Pointer(T))
1140         v := (*funcType)(unsafe.Pointer(V))
1141         if t.dotdotdot != v.dotdotdot || len(t.in) !
1142             return false
1143     }
1144     for i, typ := range t.in {
1145         if typ != v.in[i] {
1146             return false
1147         }
1148     }
1149     for i, typ := range t.out {
1150         if typ != v.out[i] {
1151             return false
1152         }
1153     }
1154     return true
1155
1156     case Interface:
1157         t := (*interfaceType)(unsafe.Pointer(T))
1158         v := (*interfaceType)(unsafe.Pointer(V))
1159         if len(t.methods) == 0 && len(v.methods) ==
1160             return true
1161     }
1162     // Might have the same methods but still
1163     // need a run time conversion.
1164     return false
1165
1166     case Map:
1167         return T.Key() == V.Key() && T.Elem() == V.E
1168
1169     case Ptr, Slice:
1170         return T.Elem() == V.Elem()
1171
1172     case Struct:
1173         t := (*structType)(unsafe.Pointer(T))
1174         v := (*structType)(unsafe.Pointer(V))
1175         if len(t.fields) != len(v.fields) {
1176             return false
1177         }
1178         for i := range t.fields {
1179             tf := &t.fields[i]

```

```
1180             vf := &v.fields[i]
1181             if tf.name != vf.name || tf.pkgPath
1182                 tf.typ != vf.typ || tf.tag !=
1183                 return false
1184             }
1185         }
1186         return true
1187     }
1188
1189     return false
1190 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/reflect/value.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package reflect
6
7 import (
8     "math"
9     "runtime"
10    "strconv"
11    "unsafe"
12 )
13
14 const bigEndian = false // can be smarter if we find a big-e
15 const ptrSize = unsafe.Sizeof((*byte)(nil))
16 const cannotSet = "cannot set value obtained from unexported
17
18 // TODO: This will have to go away when
19 // the new gc goes in.
20 func memmove(adst, asrc unsafe.Pointer, n uintptr) {
21     dst := uintptr(adst)
22     src := uintptr(asrc)
23     switch {
24     case src < dst && src+n > dst:
25         // byte copy backward
26         // careful: i is unsigned
27         for i := n; i > 0; {
28             i--
29             *(*byte)(unsafe.Pointer(dst + i)) =
30         }
31     case (n|src|dst)&(ptrSize-1) != 0:
32         // byte copy forward
33         for i := uintptr(0); i < n; i++ {
34             *(*byte)(unsafe.Pointer(dst + i)) =
35         }
36     default:
37         // word copy forward
38         for i := uintptr(0); i < n; i += ptrSize {
39             *(*uintptr)(unsafe.Pointer(dst + i)) =
40         }
41     }
42 }
43
44 // Value is the reflection interface to a Go value.
```

```

45 //
46 // Not all methods apply to all kinds of values. Restrictio
47 // if any, are noted in the documentation for each method.
48 // Use the Kind method to find out the kind of value before
49 // calling kind-specific methods. Calling a method
50 // inappropriate to the kind of type causes a run time panic
51 //
52 // The zero Value represents no value.
53 // Its IsValid method returns false, its Kind method returns
54 // its String method returns "<invalid Value>", and all othe
55 // Most functions and methods never return an invalid value.
56 // If one does, its documentation states the conditions expl
57 //
58 // A Value can be used concurrently by multiple goroutines p
59 // the underlying Go value can be used concurrently for the
60 // direct operations.
61 type Value struct {
62     // typ holds the type of the value represented by a
63     typ *commonType
64
65     // val holds the 1-word representation of the value.
66     // If flag's flagIndir bit is set, then val is a poi
67     // Otherwise val is a word holding the actual data.
68     // When the data is smaller than a word, it begins a
69     // the first byte (in the memory address sense) of v
70     // We use unsafe.Pointer so that the garbage collect
71     // knows that val could be a pointer.
72     val unsafe.Pointer
73
74     // flag holds metadata about the value.
75     // The lowest bits are flag bits:
76     //     - flagRO: obtained via unexported field, so
77     //     - flagIndir: val holds a pointer to the data
78     //     - flagAddr: v.CanAddr is true (implies flagI
79     //     - flagMethod: v is a method value.
80     // The next five bits give the Kind of the value.
81     // This repeats typ.Kind() except for method values.
82     // The remaining 23+ bits give a method number for m
83     // If flag.kind() != Func, code can assume that flag
84     // If typ.size > ptrSize, code can assume that flagI
85     flag
86
87     // A method value represents a curried method invoca
88     // like r.Read for some receiver r. The typ+val+fla
89     // the receiver r, but the flag's Kind bits say Func
90     // functions), and the top bits of the flag give the
91     // in r's type's method table.
92 }
93
94 type flag uintptr

```

```

95
96 const (
97     flagR0 flag = 1 << iota
98     flagIndir
99     flagAddr
100    flagMethod
101    flagKindShift      = iota
102    flagKindWidth      = 5 // there are 27 kinds
103    flagKindMask       flag = 1<<flagKindWidth - 1
104    flagMethodShift    = flagKindShift + flagKindWidth
105 )
106
107 func (f flag) kind() Kind {
108     return Kind((f >> flagKindShift) & flagKindMask)
109 }
110
111 // A ValueError occurs when a Value method is invoked on
112 // a Value that does not support it. Such cases are documen
113 // in the description of each method.
114 type ValueError struct {
115     Method string
116     Kind    Kind
117 }
118
119 func (e *ValueError) Error() string {
120     if e.Kind == 0 {
121         return "reflect: call of " + e.Method + " on
122     }
123     return "reflect: call of " + e.Method + " on " + e.K
124 }
125
126 // methodName returns the name of the calling method,
127 // assumed to be two stack frames above.
128 func methodName() string {
129     pc, _, _, _ := runtime.Caller(2)
130     f := runtime.FuncForPC(pc)
131     if f == nil {
132         return "unknown method"
133     }
134     return f.Name()
135 }
136
137 // An iword is the word that would be stored in an
138 // interface to represent a given value v. Specifically, if
139 // bigger than a pointer, its word is a pointer to v's data.
140 // Otherwise, its word holds the data stored
141 // in its leading bytes (so is not a pointer).
142 // Because the value sometimes holds a pointer, we use
143 // unsafe.Pointer to represent it, so that if iword appears

```

```

144 // in a struct, the garbage collector knows that might be
145 // a pointer.
146 type iword unsafe.Pointer
147
148 func (v Value) iword() iword {
149     if v.flag&flagIndir != 0 && v.typ.size <= ptrSize {
150         // Have indirect but want direct word.
151         return loadIword(v.val, v.typ.size)
152     }
153     return iword(v.val)
154 }
155
156 // loadIword loads n bytes at p from memory into an iword.
157 func loadIword(p unsafe.Pointer, n uintptr) iword {
158     // Run the copy ourselves instead of calling memmove
159     // to avoid moving w to the heap.
160     var w iword
161     switch n {
162     default:
163         panic("reflect: internal error: loadIword of
164     case 0:
165     case 1:
166         *(*uint8)(unsafe.Pointer(&w)) = *(*uint8)(p)
167     case 2:
168         *(*uint16)(unsafe.Pointer(&w)) = *(*uint16)(
169     case 3:
170         *(*[3]byte)(unsafe.Pointer(&w)) = *(*[3]byte
171     case 4:
172         *(*uint32)(unsafe.Pointer(&w)) = *(*uint32)(
173     case 5:
174         *(*[5]byte)(unsafe.Pointer(&w)) = *(*[5]byte
175     case 6:
176         *(*[6]byte)(unsafe.Pointer(&w)) = *(*[6]byte
177     case 7:
178         *(*[7]byte)(unsafe.Pointer(&w)) = *(*[7]byte
179     case 8:
180         *(*uint64)(unsafe.Pointer(&w)) = *(*uint64)(
181     }
182     return w
183 }
184
185 // storeIword stores n bytes from w into p.
186 func storeIword(p unsafe.Pointer, w iword, n uintptr) {
187     // Run the copy ourselves instead of calling memmove
188     // to avoid moving w to the heap.
189     switch n {
190     default:
191         panic("reflect: internal error: storeIword o
192     case 0:

```

```

193     case 1:
194         *(*uint8)(p) = *(*uint8)(unsafe.Pointer(&w))
195     case 2:
196         *(*uint16)(p) = *(*uint16)(unsafe.Pointer(&w))
197     case 3:
198         *(*[3]byte)(p) = *(*[3]byte)(unsafe.Pointer(&w))
199     case 4:
200         *(*uint32)(p) = *(*uint32)(unsafe.Pointer(&w))
201     case 5:
202         *(*[5]byte)(p) = *(*[5]byte)(unsafe.Pointer(&w))
203     case 6:
204         *(*[6]byte)(p) = *(*[6]byte)(unsafe.Pointer(&w))
205     case 7:
206         *(*[7]byte)(p) = *(*[7]byte)(unsafe.Pointer(&w))
207     case 8:
208         *(*uint64)(p) = *(*uint64)(unsafe.Pointer(&w))
209     }
210 }
211
212 // emptyInterface is the header for an interface{} value.
213 type emptyInterface struct {
214     typ *runtimeType
215     word iword
216 }
217
218 // nonEmptyInterface is the header for a interface value with
219 type nonEmptyInterface struct {
220     // see ../runtime/iface.c://Itab
221     itab *struct {
222         ityp *runtimeType // static interface type
223         typ *runtimeType // dynamic concrete type
224         link unsafe.Pointer
225         bad int32
226         unused int32
227         fun [100000]unsafe.Pointer // method table
228     }
229     word iword
230 }
231
232 // mustBe panics if f's kind is not expected.
233 // Making this a method on flag instead of on Value
234 // (and embedding flag in Value) means that we can write
235 // the very clear v.mustBe(Bool) and have it compile into
236 // v.flag.mustBe(Bool), which will only bother to copy the
237 // single important word for the receiver.
238 func (f flag) mustBe(expected Kind) {
239     k := f.kind()
240     if k != expected {
241         panic(&ValueError{methodName(), k})
242     }

```

```

243 }
244
245 // mustBeExported panics if f records that the value was obt
246 // an unexported field.
247 func (f flag) mustBeExported() {
248     if f == 0 {
249         panic(&ValueError{methodName(), 0})
250     }
251     if f&flagRO != 0 {
252         panic(methodName() + " using value obtained
253     }
254 }
255
256 // mustBeAssignable panics if f records that the value is no
257 // which is to say that either it was obtained using an unex
258 // or it is not addressable.
259 func (f flag) mustBeAssignable() {
260     if f == 0 {
261         panic(&ValueError{methodName(), Invalid})
262     }
263     // Assignable if addressable and not read-only.
264     if f&flagRO != 0 {
265         panic(methodName() + " using value obtained
266     }
267     if f&flagAddr == 0 {
268         panic(methodName() + " using unaddressable v
269     }
270 }
271
272 // Addr returns a pointer value representing the address of
273 // It panics if CanAddr() returns false.
274 // Addr is typically used to obtain a pointer to a struct fi
275 // or slice element in order to call a method that requires
276 // pointer receiver.
277 func (v Value) Addr() Value {
278     if v.flag&flagAddr == 0 {
279         panic("reflect.Value.Addr of unaddressable v
280     }
281     return Value{v.typ.ptrTo(), v.val, (v.flag & flagRO)
282 }
283
284 // Bool returns v's underlying value.
285 // It panics if v's kind is not Bool.
286 func (v Value) Bool() bool {
287     v.mustBe(Bool)
288     if v.flag&flagIndir != 0 {
289         return *(*bool)(v.val)
290     }
291     return *(*bool)(unsafe.Pointer(&v.val))

```

```

292 }
293
294 // Bytes returns v's underlying value.
295 // It panics if v's underlying value is not a slice of bytes
296 func (v Value) Bytes() []byte {
297     v.mustBe(Slice)
298     if v.typ.Elem().Kind() != Uint8 {
299         panic("reflect.Value.Bytes of non-byte slice")
300     }
301     // Slice is always bigger than a word; assume flagIn
302     return *(*[]byte)(v.val)
303 }
304
305 // CanAddr returns true if the value's address can be obtain
306 // Such values are called addressable. A value is addressab
307 // an element of a slice, an element of an addressable array
308 // a field of an addressable struct, or the result of derefe
309 // If CanAddr returns false, calling Addr will panic.
310 func (v Value) CanAddr() bool {
311     return v.flag&flagAddr != 0
312 }
313
314 // CanSet returns true if the value of v can be changed.
315 // A Value can be changed only if it is addressable and was
316 // obtained by the use of unexported struct fields.
317 // If CanSet returns false, calling Set or any type-specific
318 // setter (e.g., SetBool, SetInt64) will panic.
319 func (v Value) CanSet() bool {
320     return v.flag&(flagAddr|flagRO) == flagAddr
321 }
322
323 // Call calls the function v with the input arguments in.
324 // For example, if len(in) == 3, v.Call(in) represents the G
325 // Call panics if v's Kind is not Func.
326 // It returns the output results as Values.
327 // As in Go, each input argument must be assignable to the
328 // type of the function's corresponding input parameter.
329 // If v is a variadic function, Call creates the variadic sl
330 // itself, copying in the corresponding values.
331 func (v Value) Call(in []Value) []Value {
332     v.mustBe(Func)
333     v.mustBeExported()
334     return v.call("Call", in)
335 }
336
337 // CallSlice calls the variadic function v with the input ar
338 // assigning the slice in[len(in)-1] to v's final variadic a
339 // For example, if len(in) == 3, v.Call(in) represents the G
340 // Call panics if v's Kind is not Func or if v is not variad

```

```

341 // It returns the output results as Values.
342 // As in Go, each input argument must be assignable to the
343 // type of the function's corresponding input parameter.
344 func (v Value) CallSlice(in []Value) []Value {
345     v.mustBe(Func)
346     v.mustBeExported()
347     return v.call("CallSlice", in)
348 }
349
350 func (v Value) call(method string, in []Value) []Value {
351     // Get function pointer, type.
352     t := v.typ
353     var (
354         fn unsafe.Pointer
355         rcvr iword
356     )
357     if v.flag&flagMethod != 0 {
358         i := int(v.flag) >> flagMethodShift
359         if v.typ.Kind() == Interface {
360             tt := (*interfaceType)(unsafe.Pointer)
361             if i < 0 || i >= len(tt.methods) {
362                 panic("reflect: broken Value")
363             }
364             m := &tt.methods[i]
365             if m.pkgPath != nil {
366                 panic(method + " of unexport")
367             }
368             t = toCommonType(m.typ)
369             iface := (*nonEmptyInterface)(v.val)
370             if iface.itab == nil {
371                 panic(method + " of method c")
372             }
373             fn = iface.itab.fun[i]
374             rcvr = iface.word
375         } else {
376             ut := v.typ.uncommon()
377             if ut == nil || i < 0 || i >= len(ut) {
378                 panic("reflect: broken Value")
379             }
380             m := &ut.methods[i]
381             if m.pkgPath != nil {
382                 panic(method + " of unexport")
383             }
384             fn = m.ifn
385             t = toCommonType(m.mtyp)
386             rcvr = v.iword()
387         }
388     } else if v.flag&flagIndir != 0 {
389         fn = *(*unsafe.Pointer)(v.val)
390     } else {

```

```

391         fn = v.val
392     }
393
394     if fn == nil {
395         panic("reflect.Value.Call: call of nil funct
396     }
397
398     isSlice := method == "CallSlice"
399     n := t.NumIn()
400     if isSlice {
401         if !t.IsVariadic() {
402             panic("reflect: CallSlice of non-var
403         }
404         if len(in) < n {
405             panic("reflect: CallSlice with too f
406         }
407         if len(in) > n {
408             panic("reflect: CallSlice with too m
409         }
410     } else {
411         if t.IsVariadic() {
412             n--
413         }
414         if len(in) < n {
415             panic("reflect: Call with too few in
416         }
417         if !t.IsVariadic() && len(in) > n {
418             panic("reflect: Call with too many i
419         }
420     }
421     for _, x := range in {
422         if x.Kind() == Invalid {
423             panic("reflect: " + method + " using
424         }
425     }
426     for i := 0; i < n; i++ {
427         if xt, targ := in[i].Type(), t.In(i); !xt.As
428             panic("reflect: " + method + " using
429         }
430     }
431     if !isSlice && t.IsVariadic() {
432         // prepare slice for remaining values
433         m := len(in) - n
434         slice := MakeSlice(t.In(n), m, m)
435         elem := t.In(n).Elem()
436         for i := 0; i < m; i++ {
437             x := in[n+i]
438             if xt := x.Type(); !xt.AssignableTo(
439                 panic("reflect: cannot use "

```

```

440         }
441         slice.Index(i).Set(x)
442     }
443     origIn := in
444     in = make([]Value, n+1)
445     copy(in[:n], origIn)
446     in[n] = slice
447 }
448
449     nin := len(in)
450     if nin != t.NumIn() {
451         panic("reflect.Value.Call: wrong argument co
452     }
453     nout := t.NumOut()
454
455     // Compute arg size & allocate.
456     // This computation is 5g/6g/8g-dependent
457     // and probably wrong for gccgo, but so
458     // is most of this function.
459     size := uintptr(0)
460     if v.flag&flagMethod != 0 {
461         // extra word for receiver interface word
462         size += ptrSize
463     }
464     for i := 0; i < nin; i++ {
465         tv := t.In(i)
466         a := uintptr(tv.Align())
467         size = (size + a - 1) &^ (a - 1)
468         size += tv.Size()
469     }
470     size = (size + ptrSize - 1) &^ (ptrSize - 1)
471     for i := 0; i < nout; i++ {
472         tv := t.Out(i)
473         a := uintptr(tv.Align())
474         size = (size + a - 1) &^ (a - 1)
475         size += tv.Size()
476     }
477
478     // size must be > 0 in order for &args[0] to be vali
479     // the argument copying is going to round it up to
480     // a multiple of ptrSize anyway, so make it ptrSize
481     if size < ptrSize {
482         size = ptrSize
483     }
484
485     // round to pointer size
486     size = (size + ptrSize - 1) &^ (ptrSize - 1)
487
488     // Copy into args.

```

```

489         //
490         // TODO(rsc): revisit when reference counting happen
491         // The values are holding up the in references for u
492         // but something must be done for the out references
493         // For now make everything look like a pointer by pr
494         // to allocate a []*int.
495         args := make([]*int, size/ptrSize)
496         ptr := uintptr(unsafe.Pointer(&args[0]))
497         off := uintptr(0)
498         if v.flag&flagMethod != 0 {
499             // Hard-wired first argument.
500             *(*iword)(unsafe.Pointer(ptr)) = rcvr
501             off = ptrSize
502         }
503         for i, v := range in {
504             v.mustBeExported()
505             targ := t.In(i).(*commonType)
506             a := uintptr(targ.align)
507             off = (off + a - 1) &^ (a - 1)
508             n := targ.size
509             addr := unsafe.Pointer(ptr + off)
510             v = v.assignTo("reflect.Value.Call", targ, (
511                 if v.flag&flagIndir == 0 {
512                     storeIword(addr, iword(v.val), n)
513                 } else {
514                     memmove(addr, v.val, n)
515                 }
516                 off += n
517             )
518             off = (off + ptrSize - 1) &^ (ptrSize - 1)
519
520             // Call.
521             call(fn, unsafe.Pointer(ptr), uint32(size))
522
523             // Copy return values out of args.
524             //
525             // TODO(rsc): revisit like above.
526             ret := make([]Value, nout)
527             for i := 0; i < nout; i++ {
528                 tv := t.Out(i)
529                 a := uintptr(tv.Align())
530                 off = (off + a - 1) &^ (a - 1)
531                 fl := flagIndir | flag(tv.Kind())<<flagKinds
532                 ret[i] = Value{tv.common(), unsafe.Pointer(p
533                     off += tv.Size()
534             }
535
536             return ret
537     }
538

```

```

539 // Cap returns v's capacity.
540 // It panics if v's Kind is not Array, Chan, or Slice.
541 func (v Value) Cap() int {
542     k := v.kind()
543     switch k {
544     case Array:
545         return v.typ.Len()
546     case Chan:
547         return int(chancap(v.iword()))
548     case Slice:
549         // Slice is always bigger than a word; assum
550         return (*SliceHeader)(v.val).Cap
551     }
552     panic(&ValueError{"reflect.Value.Cap", k})
553 }
554
555 // Close closes the channel v.
556 // It panics if v's Kind is not Chan.
557 func (v Value) Close() {
558     v.mustBe(Chan)
559     v.mustBeExported()
560     chanclose(v.iword())
561 }
562
563 // Complex returns v's underlying value, as a complex128.
564 // It panics if v's Kind is not Complex64 or Complex128
565 func (v Value) Complex() complex128 {
566     k := v.kind()
567     switch k {
568     case Complex64:
569         if v.flag&flagIndir != 0 {
570             return complex128>(*complex64)(v.va
571         }
572         return complex128>(*complex64)(unsafe.Point
573     case Complex128:
574         // complex128 is always bigger than a word;
575         return *(*complex128)(v.val)
576     }
577     panic(&ValueError{"reflect.Value.Complex", k})
578 }
579
580 // Elem returns the value that the interface v contains
581 // or that the pointer v points to.
582 // It panics if v's Kind is not Interface or Ptr.
583 // It returns the zero Value if v is nil.
584 func (v Value) Elem() Value {
585     k := v.kind()
586     switch k {
587     case Interface:

```

```

588     var (
589         typ *commonType
590         val unsafe.Pointer
591     )
592     if v.typ.NumMethod() == 0 {
593         eface := (*emptyInterface)(v.val)
594         if eface.typ == nil {
595             // nil interface value
596             return Value{}
597         }
598         typ = toCommonType(eface.typ)
599         val = unsafe.Pointer(eface.word)
600     } else {
601         iface := (*nonEmptyInterface)(v.val)
602         if iface.itab == nil {
603             // nil interface value
604             return Value{}
605         }
606         typ = toCommonType(iface.itab.typ)
607         val = unsafe.Pointer(iface.word)
608     }
609     fl := v.flag & flagRO
610     fl |= flag(typ.Kind()) << flagKindShift
611     if typ.size > ptrSize {
612         fl |= flagIndir
613     }
614     return Value{typ, val, fl}
615
616     case Ptr:
617         val := v.val
618         if v.flag&flagIndir != 0 {
619             val = *(*unsafe.Pointer)(val)
620         }
621         // The returned value's address is v's value
622         if val == nil {
623             return Value{}
624         }
625         tt := (*ptrType)(unsafe.Pointer(v.typ))
626         typ := toCommonType(tt.elem)
627         fl := v.flag&flagRO | flagIndir | flagAddr
628         fl |= flag(typ.Kind()) << flagKindShift
629         return Value{typ, val, fl}
630     }
631     panic(&ValueError{"reflect.Value.Elem", k})
632 }
633
634 // Field returns the i'th field of the struct v.
635 // It panics if v's Kind is not Struct or i is out of range.
636 func (v Value) Field(i int) Value {

```

```

637     v.mustBe(Struct)
638     tt := (*structType)(unsafe.Pointer(v.typ))
639     if i < 0 || i >= len(tt.fields) {
640         panic("reflect: Field index out of range")
641     }
642     field := &tt.fields[i]
643     typ := toCommonType(field.typ)
644
645     // Inherit permission bits from v.
646     fl := v.flag & (flagRO | flagIndir | flagAddr)
647     // Using an unexported field forces flagRO.
648     if field.pkgPath != nil {
649         fl |= flagRO
650     }
651     fl |= flag(typ.Kind()) << flagKindShift
652
653     var val unsafe.Pointer
654     switch {
655     case fl&flagIndir != 0:
656         // Indirect. Just bump pointer.
657         val = unsafe.Pointer(uintptr(v.val) + field.
658     case bigEndian:
659         // Direct. Discard leading bytes.
660         val = unsafe.Pointer(uintptr(v.val) << (fiel
661     default:
662         // Direct. Discard leading bytes.
663         val = unsafe.Pointer(uintptr(v.val) >> (fiel
664     }
665
666     return Value{typ, val, fl}
667 }
668
669 // FieldByIndex returns the nested field corresponding to in
670 // It panics if v's Kind is not struct.
671 func (v Value) FieldByIndex(index []int) Value {
672     v.mustBe(Struct)
673     for i, x := range index {
674         if i > 0 {
675             if v.Kind() == Ptr && v.Elem().Kind(
676                 v = v.Elem()
677             }
678         }
679         v = v.Field(x)
680     }
681     return v
682 }
683
684 // FieldByName returns the struct field with the given name.
685 // It returns the zero Value if no field was found.
686 // It panics if v's Kind is not struct.

```

```

687 func (v Value) FieldByName(name string) Value {
688     v.mustBe(Struct)
689     if f, ok := v.typ.FieldByName(name); ok {
690         return v.FieldByIndex(f.Index)
691     }
692     return Value{}
693 }
694
695 // FieldByNameFunc returns the struct field with a name
696 // that satisfies the match function.
697 // It panics if v's Kind is not struct.
698 // It returns the zero Value if no field was found.
699 func (v Value) FieldByNameFunc(match func(string) bool) Value {
700     v.mustBe(Struct)
701     if f, ok := v.typ.FieldByNameFunc(match); ok {
702         return v.FieldByIndex(f.Index)
703     }
704     return Value{}
705 }
706
707 // Float returns v's underlying value, as a float64.
708 // It panics if v's Kind is not Float32 or Float64
709 func (v Value) Float() float64 {
710     k := v.kind()
711     switch k {
712     case Float32:
713         if v.flag&flagIndir != 0 {
714             return float64>(*float32)(v.val)
715         }
716         return float64>(*float32)(unsafe.Pointer(&v.val))
717     case Float64:
718         if v.flag&flagIndir != 0 {
719             return *(*float64)(v.val)
720         }
721         return *(*float64)(unsafe.Pointer(&v.val))
722     }
723     panic(&ValueError{"reflect.Value.Float", k})
724 }
725
726 // Index returns v's i'th element.
727 // It panics if v's Kind is not Array or Slice or i is out of bounds.
728 func (v Value) Index(i int) Value {
729     k := v.kind()
730     switch k {
731     case Array:
732         tt := (*arrayType)(unsafe.Pointer(v.typ))
733         if i < 0 || i > int(tt.len) {
734             panic("reflect: array index out of bounds")
735         }

```

```

736         typ := toCommonType(tt.elem)
737         fl := v.flag & (flagRO | flagIndir | flagAdd
738         fl |= flag(typ.Kind()) << flagKindShift
739         offset := uintptr(i) * typ.size
740
741         var val unsafe.Pointer
742         switch {
743         case fl&flagIndir != 0:
744             // Indirect. Just bump pointer.
745             val = unsafe.Pointer(uintptr(v.val)
746         case bigEndian:
747             // Direct. Discard leading bytes.
748             val = unsafe.Pointer(uintptr(v.val)
749         default:
750             // Direct. Discard leading bytes.
751             val = unsafe.Pointer(uintptr(v.val)
752         }
753         return Value{typ, val, fl}
754
755     case Slice:
756         // Element flag same as Elem of Ptr.
757         // Addressable, indirect, possibly read-only
758         fl := flagAddr | flagIndir | v.flag&flagRO
759         s := (*SliceHeader)(v.val)
760         if i < 0 || i >= s.Len {
761             panic("reflect: slice index out of r
762         }
763         tt := (*sliceType)(unsafe.Pointer(v.typ))
764         typ := toCommonType(tt.elem)
765         fl |= flag(typ.Kind()) << flagKindShift
766         val := unsafe.Pointer(s.Data + uintptr(i)*ty
767         return Value{typ, val, fl}
768     }
769     panic(&ValueError{"reflect.Value.Index", k})
770 }
771
772 // Int returns v's underlying value, as an int64.
773 // It panics if v's Kind is not Int, Int8, Int16, Int32, or
774 func (v Value) Int() int64 {
775     k := v.kind()
776     var p unsafe.Pointer
777     if v.flag&flagIndir != 0 {
778         p = v.val
779     } else {
780         // The escape analysis is good enough that &
781         // does not trigger a heap allocation.
782         p = unsafe.Pointer(&v.val)
783     }
784     switch k {

```

```

785     case Int:
786         return int64>(*int)(p)
787     case Int8:
788         return int64>(*int8)(p)
789     case Int16:
790         return int64>(*int16)(p)
791     case Int32:
792         return int64>(*int32)(p)
793     case Int64:
794         return int64>(*int64)(p)
795     }
796     panic(&ValueError{"reflect.Value.Int", k})
797 }
798
799 // CanInterface returns true if Interface can be used without
800 func (v Value) CanInterface() bool {
801     if v.flag == 0 {
802         panic(&ValueError{"reflect.Value.CanInterface", k})
803     }
804     return v.flag&(flagMethod|flagRO) == 0
805 }
806
807 // Interface returns v's current value as an interface{}.
808 // It is equivalent to:
809 //     var i interface{} = (v's underlying value)
810 // If v is a method obtained by invoking Value.Method
811 // (as opposed to Type.Method), Interface cannot return an
812 // interface value, so it panics.
813 // It also panics if the Value was obtained by accessing
814 // unexported struct fields.
815 func (v Value) Interface() (i interface{}) {
816     return valueInterface(v, true)
817 }
818
819 func valueInterface(v Value, safe bool) interface{} {
820     if v.flag == 0 {
821         panic(&ValueError{"reflect.Value.Interface", k})
822     }
823     if v.flag&flagMethod != 0 {
824         panic("reflect.Value.Interface: cannot create interface from method")
825     }
826
827     if safe && v.flag&flagRO != 0 {
828         // Do not allow access to unexported values
829         // because they might be pointers that should
830         // be writable or methods or function that should
831         // be callable.
832         panic("reflect.Value.Interface: cannot return interface for unexported value or method")
833     }
834     k := v.kind()

```

```

835     if k == Interface {
836         // Special case: return the element inside t
837         // Empty interface has one layout, all inter
838         // methods have a second layout.
839         if v.NumMethod() == 0 {
840             return *(*interface{})(v.val)
841         }
842         return *(*interface {
843             M()
844         })(v.val)
845     }
846
847     // Non-interface value.
848     var eface emptyInterface
849     eface.typ = v.typ.runtimeType()
850     eface.word = v.iword()
851
852     if v.flag&flagIndir != 0 && v.typ.size > ptrSize {
853         // eface.word is a pointer to the actual dat
854         // which might be changed. We need to retur
855         // a pointer to unchanging data, so make a c
856         ptr := unsafe_New(v.typ)
857         memmove(ptr, unsafe.Pointer(eface.word), v.t
858         eface.word = iword(ptr)
859     }
860
861     return *(*interface{})(unsafe.Pointer(&eface))
862 }
863
864 // InterfaceData returns the interface v's value as a uintptr
865 // It panics if v's Kind is not Interface.
866 func (v Value) InterfaceData() [2]uintptr {
867     v.mustBe(Interface)
868     // We treat this as a read operation, so we allow
869     // it even for unexported data, because the caller
870     // has to import "unsafe" to turn it into something
871     // that can be abused.
872     // Interface value is always bigger than a word; ass
873     return *(*[2]uintptr)(v.val)
874 }
875
876 // IsNil returns true if v is a nil value.
877 // It panics if v's Kind is not Chan, Func, Interface, Map,
878 func (v Value) IsNil() bool {
879     k := v.kind()
880     switch k {
881     case Chan, Func, Map, Ptr:
882         if v.flag&flagMethod != 0 {
883             panic("reflect: IsNil of method Valu

```

```

884         }
885         ptr := v.val
886         if v.flag&flagIndir != 0 {
887             ptr = *(*unsafe.Pointer)(ptr)
888         }
889         return ptr == nil
890     case Interface, Slice:
891         // Both interface and slice are nil if first
892         // Both are always bigger than a word; assum
893         return *(*unsafe.Pointer)(v.val) == nil
894     }
895     panic(&ValueError{"reflect.Value.IsNil", k})
896 }
897
898 // IsValid returns true if v represents a value.
899 // It returns false if v is the zero Value.
900 // If IsValid returns false, all other methods except String
901 // Most functions and methods never return an invalid value.
902 // If one does, its documentation states the conditions expl
903 func (v Value) IsValid() bool {
904     return v.flag != 0
905 }
906
907 // Kind returns v's Kind.
908 // If v is the zero Value (IsValid returns false), Kind retu
909 func (v Value) Kind() Kind {
910     return v.kind()
911 }
912
913 // Len returns v's length.
914 // It panics if v's Kind is not Array, Chan, Map, Slice, or
915 func (v Value) Len() int {
916     k := v.kind()
917     switch k {
918     case Array:
919         tt := (*arrayType)(unsafe.Pointer(v.typ))
920         return int(tt.len)
921     case Chan:
922         return int(chanlen(v.iword()))
923     case Map:
924         return int(maplen(v.iword()))
925     case Slice:
926         // Slice is bigger than a word; assume flagI
927         return (*SliceHeader)(v.val).Len
928     case String:
929         // String is bigger than a word; assume flag
930         return (*StringHeader)(v.val).Len
931     }
932     panic(&ValueError{"reflect.Value.Len", k})

```

```

933 }
934
935 // MapIndex returns the value associated with key in the map
936 // It panics if v's Kind is not Map.
937 // It returns the zero Value if key is not found in the map
938 // As in Go, the key's value must be assignable to the map's
939 func (v Value) MapIndex(key Value) Value {
940     v.mustBe(Map)
941     tt := (*mapType)(unsafe.Pointer(v.typ))
942
943     // Do not require key to be exported, so that DeepEq
944     // and other programs can use all the keys returned
945     // MapKeys as arguments to MapIndex. If either the
946     // or the key is unexported, though, the result will
947     // considered unexported. This is consistent with t
948     // behavior for structs, which allow read but not wr
949     // of unexported fields.
950     key = key.assignTo("reflect.Value.MapIndex", toCommo
951
952     word, ok := mapaccess(v.typ.runtimeType(), v.iword())
953     if !ok {
954         return Value{}
955     }
956     typ := toCommonType(tt.elem)
957     fl := (v.flag | key.flag) & flagRO
958     if typ.size > ptrSize {
959         fl |= flagIndir
960     }
961     fl |= flag(typ.Kind()) << flagKindShift
962     return Value{typ, unsafe.Pointer(word), fl}
963 }
964
965 // MapKeys returns a slice containing all the keys present i
966 // in unspecified order.
967 // It panics if v's Kind is not Map.
968 // It returns an empty slice if v represents a nil map.
969 func (v Value) MapKeys() []Value {
970     v.mustBe(Map)
971     tt := (*mapType)(unsafe.Pointer(v.typ))
972     keyType := toCommonType(tt.key)
973
974     fl := v.flag & flagRO
975     fl |= flag(keyType.Kind()) << flagKindShift
976     if keyType.size > ptrSize {
977         fl |= flagIndir
978     }
979
980     m := v.iword()
981     mlen := int32(0)
982     if m != nil {

```

```

983         mlen = maplen(m)
984     }
985     it := mapiterinit(v.typ.runtimeType(), m)
986     a := make([]Value, mlen)
987     var i int
988     for i = 0; i < len(a); i++ {
989         keyword, ok := mapiterkey(it)
990         if !ok {
991             break
992         }
993         a[i] = Value{keyType, unsafe.Pointer(keyword
994             mapiternext(it)
995         }
996     return a[:i]
997 }
998
999 // Method returns a function value corresponding to v's i'th
1000 // The arguments to a Call on the returned function should n
1001 // a receiver; the returned function will always use v as th
1002 // Method panics if i is out of range.
1003 func (v Value) Method(i int) Value {
1004     if v.typ == nil {
1005         panic(&ValueError{"reflect.Value.Method", In
1006     }
1007     if v.flag&flagMethod != 0 || i < 0 || i >= v.typ.Num
1008         panic("reflect: Method index out of range")
1009     }
1010     fl := v.flag & (flagRO | flagAddr | flagIndir)
1011     fl |= flag(Func) << flagKindShift
1012     fl |= flag(i)<<flagMethodShift | flagMethod
1013     return Value{v.typ, v.val, fl}
1014 }
1015
1016 // NumMethod returns the number of methods in the value's me
1017 func (v Value) NumMethod() int {
1018     if v.typ == nil {
1019         panic(&ValueError{"reflect.Value.NumMethod",
1020     }
1021     if v.flag&flagMethod != 0 {
1022         return 0
1023     }
1024     return v.typ.NumMethod()
1025 }
1026
1027 // MethodByName returns a function value corresponding to th
1028 // of v with the given name.
1029 // The arguments to a Call on the returned function should n
1030 // a receiver; the returned function will always use v as th
1031 // It returns the zero Value if no method was found.

```

```

1032 func (v Value) MethodByName(name string) Value {
1033     if v.typ == nil {
1034         panic(&ValueError{"reflect.Value.MethodByName
1035     }
1036     if v.flag&flagMethod != 0 {
1037         return Value{}
1038     }
1039     m, ok := v.typ.MethodByName(name)
1040     if !ok {
1041         return Value{}
1042     }
1043     return v.Method(m.Index)
1044 }
1045
1046 // NumField returns the number of fields in the struct v.
1047 // It panics if v's Kind is not Struct.
1048 func (v Value) NumField() int {
1049     v.mustBe(Struct)
1050     tt := (*structType)(unsafe.Pointer(v.typ))
1051     return len(tt.fields)
1052 }
1053
1054 // OverflowComplex returns true if the complex128 x cannot b
1055 // It panics if v's Kind is not Complex64 or Complex128.
1056 func (v Value) OverflowComplex(x complex128) bool {
1057     k := v.kind()
1058     switch k {
1059     case Complex64:
1060         return overflowFloat32(real(x)) || overflowF
1061     case Complex128:
1062         return false
1063     }
1064     panic(&ValueError{"reflect.Value.OverflowComplex", k
1065 }
1066
1067 // OverflowFloat returns true if the float64 x cannot be rep
1068 // It panics if v's Kind is not Float32 or Float64.
1069 func (v Value) OverflowFloat(x float64) bool {
1070     k := v.kind()
1071     switch k {
1072     case Float32:
1073         return overflowFloat32(x)
1074     case Float64:
1075         return false
1076     }
1077     panic(&ValueError{"reflect.Value.OverflowFloat", k})
1078 }
1079
1080 func overflowFloat32(x float64) bool {

```

```

1081         if x < 0 {
1082             x = -x
1083         }
1084         return math.MaxFloat32 <= x && x <= math.MaxFloat64
1085     }
1086
1087     // OverflowInt returns true if the int64 x cannot be represe
1088     // It panics if v's Kind is not Int, Int8, int16, Int32, or
1089     func (v Value) OverflowInt(x int64) bool {
1090         k := v.kind()
1091         switch k {
1092         case Int, Int8, Int16, Int32, Int64:
1093             bitSize := v.typ.size * 8
1094             trunc := (x << (64 - bitSize)) >> (64 - bitS
1095             return x != trunc
1096         }
1097         panic(&ValueError{"reflect.Value.OverflowInt", k})
1098     }
1099
1100     // OverflowUint returns true if the uint64 x cannot be repre
1101     // It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16
1102     func (v Value) OverflowUint(x uint64) bool {
1103         k := v.kind()
1104         switch k {
1105         case Uint, Uintptr, Uint8, Uint16, Uint32, Uint64:
1106             bitSize := v.typ.size * 8
1107             trunc := (x << (64 - bitSize)) >> (64 - bitS
1108             return x != trunc
1109         }
1110         panic(&ValueError{"reflect.Value.OverflowUint", k})
1111     }
1112
1113     // Pointer returns v's value as a uintptr.
1114     // It returns uintptr instead of unsafe.Pointer so that
1115     // code using reflect cannot obtain unsafe.Pointers
1116     // without importing the unsafe package explicitly.
1117     // It panics if v's Kind is not Chan, Func, Map, Ptr, Slice,
1118     func (v Value) Pointer() uintptr {
1119         k := v.kind()
1120         switch k {
1121         case Chan, Func, Map, Ptr, UnsafePointer:
1122             if k == Func && v.flag&flagMethod != 0 {
1123                 panic("reflect.Value.Pointer of meth
1124             }
1125             p := v.val
1126             if v.flag&flagIndir != 0 {
1127                 p = *(*unsafe.Pointer)(p)
1128             }
1129             return uintptr(p)
1130         case Slice:

```

```

1131         return (*SliceHeader)(v.val).Data
1132     }
1133     panic(&ValueError{"reflect.Value.Pointer", k})
1134 }
1135
1136 // Recv receives and returns a value from the channel v.
1137 // It panics if v's Kind is not Chan.
1138 // The receive blocks until a value is ready.
1139 // The boolean value ok is true if the value x corresponds t
1140 // on the channel, false if it is a zero value received beca
1141 func (v Value) Recv() (x Value, ok bool) {
1142     v.mustBe(Chan)
1143     v.mustBeExported()
1144     return v.recv(false)
1145 }
1146
1147 // internal recv, possibly non-blocking (nb).
1148 // v is known to be a channel.
1149 func (v Value) recv(nb bool) (val Value, ok bool) {
1150     tt := (*chanType)(unsafe.Pointer(v.typ))
1151     if ChanDir(tt.dir)&RecvDir == 0 {
1152         panic("recv on send-only channel")
1153     }
1154     word, selected, ok := chanrecv(v.typ.runtimeType(),
1155     if selected {
1156         typ := toCommonType(tt.elem)
1157         fl := flag(typ.Kind()) << flagKindShift
1158         if typ.size > ptrSize {
1159             fl |= flagIndir
1160         }
1161         val = Value{typ, unsafe.Pointer(word), fl}
1162     }
1163     return
1164 }
1165
1166 // Send sends x on the channel v.
1167 // It panics if v's kind is not Chan or if x's type is not t
1168 // As in Go, x's value must be assignable to the channel's e
1169 func (v Value) Send(x Value) {
1170     v.mustBe(Chan)
1171     v.mustBeExported()
1172     v.send(x, false)
1173 }
1174
1175 // internal send, possibly non-blocking.
1176 // v is known to be a channel.
1177 func (v Value) send(x Value, nb bool) (selected bool) {
1178     tt := (*chanType)(unsafe.Pointer(v.typ))
1179     if ChanDir(tt.dir)&SendDir == 0 {

```

```

1180         panic("send on recv-only channel")
1181     }
1182     x.mustBeExported()
1183     x = x.assignTo("reflect.Value.Send", toCommonType(tt
1184     return chansend(v.typ.runtimeType(), v.iword(), x.iw
1185 }
1186
1187 // Set assigns x to the value v.
1188 // It panics if CanSet returns false.
1189 // As in Go, x's value must be assignable to v's type.
1190 func (v Value) Set(x Value) {
1191     v.mustBeAssignable()
1192     x.mustBeExported() // do not let unexported x leak
1193     var target *interface{}
1194     if v.kind() == Interface {
1195         target = (*interface{})(v.val)
1196     }
1197     x = x.assignTo("reflect.Set", v.typ, target)
1198     if x.flag&flagIndir != 0 {
1199         memmove(v.val, x.val, v.typ.size)
1200     } else {
1201         storeIword(v.val, iword(x.val), v.typ.size)
1202     }
1203 }
1204
1205 // SetBool sets v's underlying value.
1206 // It panics if v's Kind is not Bool or if CanSet() is false
1207 func (v Value) SetBool(x bool) {
1208     v.mustBeAssignable()
1209     v.mustBe(Bool)
1210     *(*bool)(v.val) = x
1211 }
1212
1213 // SetBytes sets v's underlying value.
1214 // It panics if v's underlying value is not a slice of bytes
1215 func (v Value) SetBytes(x []byte) {
1216     v.mustBeAssignable()
1217     v.mustBe(Slice)
1218     if v.typ.Elem().Kind() != Uint8 {
1219         panic("reflect.Value.SetBytes of non-byte sl
1220     }
1221     *(*[]byte)(v.val) = x
1222 }
1223
1224 // SetComplex sets v's underlying value to x.
1225 // It panics if v's Kind is not Complex64 or Complex128, or
1226 func (v Value) SetComplex(x complex128) {
1227     v.mustBeAssignable()
1228     switch k := v.kind(); k {

```

```

1229         default:
1230             panic(&ValueError{"reflect.Value.SetComplex"
1231 case Complex64:
1232             *(*complex64)(v.val) = complex64(x)
1233 case Complex128:
1234             *(*complex128)(v.val) = x
1235     }
1236 }
1237
1238 // SetFloat sets v's underlying value to x.
1239 // It panics if v's Kind is not Float32 or Float64, or if Ca
1240 func (v Value) SetFloat(x float64) {
1241     v.mustBeAssignable()
1242     switch k := v.kind(); k {
1243     default:
1244         panic(&ValueError{"reflect.Value.SetFloat",
1245 case Float32:
1246         *(*float32)(v.val) = float32(x)
1247 case Float64:
1248         *(*float64)(v.val) = x
1249     }
1250 }
1251
1252 // SetInt sets v's underlying value to x.
1253 // It panics if v's Kind is not Int, Int8, Int16, Int32, or
1254 func (v Value) SetInt(x int64) {
1255     v.mustBeAssignable()
1256     switch k := v.kind(); k {
1257     default:
1258         panic(&ValueError{"reflect.Value.SetInt", k}
1259 case Int:
1260         *(*int)(v.val) = int(x)
1261 case Int8:
1262         *(*int8)(v.val) = int8(x)
1263 case Int16:
1264         *(*int16)(v.val) = int16(x)
1265 case Int32:
1266         *(*int32)(v.val) = int32(x)
1267 case Int64:
1268         *(*int64)(v.val) = x
1269     }
1270 }
1271
1272 // SetLen sets v's length to n.
1273 // It panics if v's Kind is not Slice or if n is negative or
1274 // greater than the capacity of the slice.
1275 func (v Value) SetLen(n int) {
1276     v.mustBeAssignable()
1277     v.mustBe(Slice)
1278     s := (*SliceHeader)(v.val)

```

```

1279         if n < 0 || n > int(s.Cap) {
1280             panic("reflect: slice length out of range in
1281         }
1282         s.Len = n
1283     }
1284
1285     // SetMapIndex sets the value associated with key in the map
1286     // It panics if v's Kind is not Map.
1287     // If val is the zero Value, SetMapIndex deletes the key fro
1288     // As in Go, key's value must be assignable to the map's key
1289     // and val's value must be assignable to the map's value typ
1290     func (v Value) SetMapIndex(key, val Value) {
1291         v.mustBe(Map)
1292         v.mustBeExported()
1293         key.mustBeExported()
1294         tt := (*mapType)(unsafe.Pointer(v.typ))
1295         key = key.assignTo("reflect.Value.SetMapIndex", toCo
1296         if val.typ != nil {
1297             val.mustBeExported()
1298             val = val.assignTo("reflect.Value.SetMapInde
1299         }
1300         mapassign(v.typ.runtimeType(), v.iword(), key.iword(
1301     }
1302
1303     // SetUint sets v's underlying value to x.
1304     // It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16
1305     func (v Value) SetUint(x uint64) {
1306         v.mustBeAssignable()
1307         switch k := v.kind(); k {
1308         default:
1309             panic(&ValueError{"reflect.Value.SetUint", k
1310         case Uint:
1311             *(*uint)(v.val) = uint(x)
1312         case Uint8:
1313             *(*uint8)(v.val) = uint8(x)
1314         case Uint16:
1315             *(*uint16)(v.val) = uint16(x)
1316         case Uint32:
1317             *(*uint32)(v.val) = uint32(x)
1318         case Uint64:
1319             *(*uint64)(v.val) = x
1320         case Uintptr:
1321             *(*uintptr)(v.val) = uintptr(x)
1322         }
1323     }
1324
1325     // SetPointer sets the unsafe.Pointer value v to x.
1326     // It panics if v's Kind is not UnsafePointer.
1327     func (v Value) SetPointer(x unsafe.Pointer) {

```

```

1328         v.mustBeAssignable()
1329         v.mustBe(UnsafePointer)
1330         *(*unsafe.Pointer)(v.val) = x
1331     }
1332
1333     // SetString sets v's underlying value to x.
1334     // It panics if v's Kind is not String or if CanSet() is fal
1335     func (v Value) SetString(x string) {
1336         v.mustBeAssignable()
1337         v.mustBe(String)
1338         *(*string)(v.val) = x
1339     }
1340
1341     // Slice returns a slice of v.
1342     // It panics if v's Kind is not Array or Slice.
1343     func (v Value) Slice(beg, end int) Value {
1344         var (
1345             cap int
1346             typ *sliceType
1347             base unsafe.Pointer
1348         )
1349         switch k := v.kind(); k {
1350         default:
1351             panic(&ValueError{"reflect.Value.Slice", k})
1352         case Array:
1353             if v.flag&flagAddr == 0 {
1354                 panic("reflect.Value.Slice: slice of
1355             }
1356             tt := (*arrayType)(unsafe.Pointer(v.typ))
1357             cap = int(tt.len)
1358             typ = (*sliceType)(unsafe.Pointer(toCommonTy
1359             base = v.val
1360         case Slice:
1361             typ = (*sliceType)(unsafe.Pointer(v.typ))
1362             s := (*SliceHeader)(v.val)
1363             base = unsafe.Pointer(s.Data)
1364             cap = s.Cap
1365         }
1366         if beg < 0 || end < beg || end > cap {
1367             panic("reflect.Value.Slice: slice index out
1368         }
1369
1370         // Declare slice so that gc can see the base pointer
1371         var x []byte
1372
1373         // Reinterpret as *SliceHeader to edit.
1374         s := (*SliceHeader)(unsafe.Pointer(&x))
1375         s.Data = uintptr(base) + uintptr(beg)*toCommonType(t
1376

```

```

1377         s.Len = end - beg
1378         s.Cap = cap - beg
1379
1380         fl := v.flag&flagRO | flagIndir | flag(Slice)<<flagK
1381         return Value{typ.common(), unsafe.Pointer(&x), fl}
1382     }
1383
1384     // String returns the string v's underlying value, as a string.
1385     // String is a special case because of Go's String method.
1386     // Unlike the other getters, it does not panic if v's Kind is Invalid.
1387     // Instead, it returns a string of the form "<T value>" where T is the type.
1388     func (v Value) String() string {
1389         switch k := v.kind(); k {
1390         case Invalid:
1391             return "<invalid Value>"
1392         case String:
1393             return *(*string)(v.val)
1394         }
1395         // If you call String on a reflect.Value of other type,
1396         // print something than to panic. Useful in debugging.
1397         return "<" + v.typ.String() + " Value>"
1398     }
1399
1400     // TryRecv attempts to receive a value from the channel v but will not
1401     // It panics if v's Kind is not Chan.
1402     // If the receive cannot finish without blocking, x is the zero value of the type.
1403     // The boolean ok is true if the value x corresponds to a send on the channel,
1404     // false if it is a zero value received because the channel was closed.
1405     func (v Value) TryRecv() (x Value, ok bool) {
1406         v.mustBe(Chan)
1407         v.mustBeExported()
1408         return v.recv(true)
1409     }
1410
1411     // TrySend attempts to send x on the channel v but will not
1412     // It panics if v's Kind is not Chan.
1413     // It returns true if the value was sent, false otherwise.
1414     // As in Go, x's value must be assignable to the channel's element type.
1415     func (v Value) TrySend(x Value) bool {
1416         v.mustBe(Chan)
1417         v.mustBeExported()
1418         return v.send(x, true)
1419     }
1420
1421     // Type returns v's type.
1422     func (v Value) Type() Type {
1423         f := v.flag
1424         if f == 0 {
1425             panic(&ValueError{"reflect.Value.Type", Invalid})
1426         }

```

```

1427     if f&flagMethod == 0 {
1428         // Easy case
1429         return v.typ
1430     }
1431
1432     // Method value.
1433     // v.typ describes the receiver, not the method type
1434     i := int(v.flag) >> flagMethodShift
1435     if v.typ.Kind() == Interface {
1436         // Method on interface.
1437         tt := (*interfaceType)(unsafe.Pointer(v.typ))
1438         if i < 0 || i >= len(tt.methods) {
1439             panic("reflect: broken Value")
1440         }
1441         m := &tt.methods[i]
1442         return toCommonType(m.typ)
1443     }
1444     // Method on concrete type.
1445     ut := v.typ.uncommon()
1446     if ut == nil || i < 0 || i >= len(ut.methods) {
1447         panic("reflect: broken Value")
1448     }
1449     m := &ut.methods[i]
1450     return toCommonType(m.mtyp)
1451 }
1452
1453 // Uint returns v's underlying value, as a uint64.
1454 // It panics if v's Kind is not Uint, Uintptr, Uint8, Uint16
1455 func (v Value) Uint() uint64 {
1456     k := v.kind()
1457     var p unsafe.Pointer
1458     if v.flag&flagIndir != 0 {
1459         p = v.val
1460     } else {
1461         // The escape analysis is good enough that &
1462         // does not trigger a heap allocation.
1463         p = unsafe.Pointer(&v.val)
1464     }
1465     switch k {
1466     case Uint:
1467         return uint64>(*uint)(p)
1468     case Uint8:
1469         return uint64>(*uint8)(p)
1470     case Uint16:
1471         return uint64>(*uint16)(p)
1472     case Uint32:
1473         return uint64>(*uint32)(p)
1474     case Uint64:
1475         return uint64>(*uint64)(p)

```

```

1476         case Uintptr:
1477             return uint64>(*uintptr)(p))
1478         }
1479         panic(&ValueError{"reflect.Value.Uint", k})
1480     }
1481
1482     // UnsafeAddr returns a pointer to v's data.
1483     // It is for advanced clients that also import the "unsafe"
1484     // It panics if v is not addressable.
1485     func (v Value) UnsafeAddr() uintptr {
1486         if v.typ == nil {
1487             panic(&ValueError{"reflect.Value.UnsafeAddr"
1488         }
1489         if v.flag&flagAddr == 0 {
1490             panic("reflect.Value.UnsafeAddr of unaddress
1491         }
1492         return uintptr(v.val)
1493     }
1494
1495     // StringHeader is the runtime representation of a string.
1496     // It cannot be used safely or portably.
1497     type StringHeader struct {
1498         Data uintptr
1499         Len   int
1500     }
1501
1502     // SliceHeader is the runtime representation of a slice.
1503     // It cannot be used safely or portably.
1504     type SliceHeader struct {
1505         Data uintptr
1506         Len   int
1507         Cap   int
1508     }
1509
1510     func typesMustMatch(what string, t1, t2 Type) {
1511         if t1 != t2 {
1512             panic(what + ": " + t1.String() + " != " + t
1513         }
1514     }
1515
1516     // grow grows the slice s so that it can hold extra more val
1517     // more capacity if needed. It also returns the old and new
1518     func grow(s Value, extra int) (Value, int, int) {
1519         i0 := s.Len()
1520         i1 := i0 + extra
1521         if i1 < i0 {
1522             panic("reflect.Append: slice overflow")
1523         }
1524         m := s.Cap()

```

```

1525     if i1 <= m {
1526         return s.Slice(0, i1), i0, i1
1527     }
1528     if m == 0 {
1529         m = extra
1530     } else {
1531         for m < i1 {
1532             if i0 < 1024 {
1533                 m += m
1534             } else {
1535                 m += m / 4
1536             }
1537         }
1538     }
1539     t := MakeSlice(s.Type(), i1, m)
1540     Copy(t, s)
1541     return t, i0, i1
1542 }
1543
1544 // Append appends the values x to a slice s and returns the
1545 // As in Go, each x's value must be assignable to the slice'
1546 func Append(s Value, x ...Value) Value {
1547     s.mustBe(Slice)
1548     s, i0, i1 := grow(s, len(x))
1549     for i, j := i0, 0; i < i1; i, j = i+1, j+1 {
1550         s.Index(i).Set(x[j])
1551     }
1552     return s
1553 }
1554
1555 // AppendSlice appends a slice t to a slice s and returns th
1556 // The slices s and t must have the same element type.
1557 func AppendSlice(s, t Value) Value {
1558     s.mustBe(Slice)
1559     t.mustBe(Slice)
1560     typesMustMatch("reflect.AppendSlice", s.Type().Elem(
1561     s, i0, i1 := grow(s, t.Len())
1562     Copy(s.Slice(i0, i1), t)
1563     return s
1564 }
1565
1566 // Copy copies the contents of src into dst until either
1567 // dst has been filled or src has been exhausted.
1568 // It returns the number of elements copied.
1569 // Dst and src each must have kind Slice or Array, and
1570 // dst and src must have the same element type.
1571 func Copy(dst, src Value) int {
1572     dk := dst.kind()
1573     if dk != Array && dk != Slice {
1574         panic(&ValueError{"reflect.Copy", dk})

```

```

1575     }
1576     if dk == Array {
1577         dst.mustBeAssignable()
1578     }
1579     dst.mustBeExported()
1580
1581     sk := src.kind()
1582     if sk != Array && sk != Slice {
1583         panic(&ValueError{"reflect.Copy", sk})
1584     }
1585     src.mustBeExported()
1586
1587     de := dst.typ.Elem()
1588     se := src.typ.Elem()
1589     typesMustMatch("reflect.Copy", de, se)
1590
1591     n := dst.Len()
1592     if sn := src.Len(); n > sn {
1593         n = sn
1594     }
1595
1596     // If sk is an in-line array, cannot take its address
1597     // Instead, copy element by element.
1598     if src.flag&flagIndir == 0 {
1599         for i := 0; i < n; i++ {
1600             dst.Index(i).Set(src.Index(i))
1601         }
1602         return n
1603     }
1604
1605     // Copy via memmove.
1606     var da, sa unsafe.Pointer
1607     if dk == Array {
1608         da = dst.val
1609     } else {
1610         da = unsafe.Pointer((*SliceHeader)(dst.val).
1611     }
1612     if sk == Array {
1613         sa = src.val
1614     } else {
1615         sa = unsafe.Pointer((*SliceHeader)(src.val).
1616     }
1617     memmove(da, sa, uintptr(n)*de.Size())
1618     return n
1619 }
1620
1621 /*
1622  * constructors
1623  */

```

```

1624
1625 // implemented in package runtime
1626 func unsafe_New(Type) unsafe.Pointer
1627 func unsafe_NewArray(Type, int) unsafe.Pointer
1628
1629 // MakeSlice creates a new zero-initialized slice value
1630 // for the specified slice type, length, and capacity.
1631 func MakeSlice(typ Type, len, cap int) Value {
1632     if typ.Kind() != Slice {
1633         panic("reflect.MakeSlice of non-slice type")
1634     }
1635     if len < 0 {
1636         panic("reflect.MakeSlice: negative len")
1637     }
1638     if cap < 0 {
1639         panic("reflect.MakeSlice: negative cap")
1640     }
1641     if len > cap {
1642         panic("reflect.MakeSlice: len > cap")
1643     }
1644
1645     // Declare slice so that gc can see the base pointer
1646     var x []byte
1647
1648     // Reinterpret as *SliceHeader to edit.
1649     s := (*SliceHeader)(unsafe.Pointer(&x))
1650     s.Data = uintptr(unsafe_NewArray(typ.Elem(), cap))
1651     s.Len = len
1652     s.Cap = cap
1653
1654     return Value{typ.common(), unsafe.Pointer(&x), flagI
1655 }
1656
1657 // MakeChan creates a new channel with the specified type an
1658 func MakeChan(typ Type, buffer int) Value {
1659     if typ.Kind() != Chan {
1660         panic("reflect.MakeChan of non-chan type")
1661     }
1662     if buffer < 0 {
1663         panic("reflect.MakeChan: negative buffer siz
1664     }
1665     if typ.ChanDir() != BothDir {
1666         panic("reflect.MakeChan: unidirectional chan
1667     }
1668     ch := makechan(typ.runtimeType(), uint32(buffer))
1669     return Value{typ.common(), unsafe.Pointer(ch), flag(
1670 }
1671
1672 // MakeMap creates a new map of the specified type.

```

```

1673 func MakeMap(typ Type) Value {
1674     if typ.Kind() != Map {
1675         panic("reflect.MakeMap of non-map type")
1676     }
1677     m := makemap(typ.runtimeType())
1678     return Value{typ.common(), unsafe.Pointer(m), flag(M
1679 }
1680
1681 // Indirect returns the value that v points to.
1682 // If v is a nil pointer, Indirect returns a zero Value.
1683 // If v is not a pointer, Indirect returns v.
1684 func Indirect(v Value) Value {
1685     if v.Kind() != Ptr {
1686         return v
1687     }
1688     return v.Elem()
1689 }
1690
1691 // ValueOf returns a new Value initialized to the concrete v
1692 // stored in the interface i. ValueOf(nil) returns the zero
1693 func ValueOf(i interface{}) Value {
1694     if i == nil {
1695         return Value{}
1696     }
1697
1698     // TODO(rsc): Eliminate this terrible hack.
1699     // In the call to packValue, eface.typ doesn't escap
1700     // and eface.word is an integer. So it looks like
1701     // i (= eface) doesn't escape. But really it does,
1702     // because eface.word is actually a pointer.
1703     escapes(i)
1704
1705     // For an interface value with the noAddr bit set,
1706     // the representation is identical to an empty inter
1707     eface := *(*emptyInterface)(unsafe.Pointer(&i))
1708     typ := toCommonType(eface.typ)
1709     fl := flag(typ.Kind()) << flagKindShift
1710     if typ.size > ptrSize {
1711         fl |= flagIndir
1712     }
1713     return Value{typ, unsafe.Pointer(eface.word), fl}
1714 }
1715
1716 // Zero returns a Value representing a zero value for the sp
1717 // The result is different from the zero value of the Value
1718 // which represents no value at all.
1719 // For example, Zero(.TypeOf(42)) returns a Value with Kind I
1720 func Zero(typ Type) Value {
1721     if typ == nil {
1722         panic("reflect: Zero(nil)")

```

```

1723     }
1724     t := typ.common()
1725     fl := flag(t.Kind()) << flagKindShift
1726     if t.size <= ptrSize {
1727         return Value{t, nil, fl}
1728     }
1729     return Value{t, unsafe_New(typ), fl | flagIndir}
1730 }
1731
1732 // New returns a Value representing a pointer to a new zero
1733 // for the specified type. That is, the returned Value's Ty
1734 func New(typ Type) Value {
1735     if typ == nil {
1736         panic("reflect: New(nil)")
1737     }
1738     ptr := unsafe_New(typ)
1739     fl := flag(Ptr) << flagKindShift
1740     return Value{typ.common().ptrTo(), ptr, fl}
1741 }
1742
1743 // NewAt returns a Value representing a pointer to a value o
1744 // specified type, using p as that pointer.
1745 func NewAt(typ Type, p unsafe.Pointer) Value {
1746     fl := flag(Ptr) << flagKindShift
1747     return Value{typ.common().ptrTo(), p, fl}
1748 }
1749
1750 // assignTo returns a value v that can be assigned directly
1751 // It panics if v is not assignable to typ.
1752 // For a conversion to an interface type, target is a sugges
1753 func (v Value) assignTo(context string, dst *commonType, tar
1754     if v.flag&flagMethod != 0 {
1755         panic(context + ": cannot assign method valu
1756     }
1757
1758     switch {
1759     case directlyAssignable(dst, v.typ):
1760         // Overwrite type so that they match.
1761         // Same memory layout, so no harm done.
1762         v.typ = dst
1763         fl := v.flag & (flagRO | flagAddr | flagIndi
1764         fl |= flag(dst.Kind()) << flagKindShift
1765         return Value{dst, v.val, fl}
1766
1767     case implements(dst, v.typ):
1768         if target == nil {
1769             target = new(interface{})
1770         }
1771         x := valueInterface(v, false)

```

```

1772         if dst.NumMethod() == 0 {
1773             *target = x
1774         } else {
1775             ifaceE2I(dst.runtimeType(), x, unsafe.Pointer(target))
1776         }
1777         return Value{dst, unsafe.Pointer(target), false}
1778     }
1779
1780     // Failed.
1781     panic(context + ": value of type " + v.typ.String())
1782 }
1783
1784 // implemented in ../pkg/runtime
1785 func chanscap(ch iword) int32
1786 func chanclose(ch iword)
1787 func chanlen(ch iword) int32
1788 func chanrecv(t *runtimeType, ch iword, nb bool) (val iword,
1789 func chansend(t *runtimeType, ch iword, val iword, nb bool)
1790
1791 func makechan(typ *runtimeType, size uint32) (ch iword)
1792 func makemap(t *runtimeType) (m iword)
1793 func mapaccess(t *runtimeType, m iword, key iword) (val iword,
1794 func mapassign(t *runtimeType, m iword, key, val iword, ok bool)
1795 func mapiterinit(t *runtimeType, m iword) *byte
1796 func mapiterkey(it *byte) (key iword, ok bool)
1797 func mapiternext(it *byte)
1798 func maplen(m iword) int32
1799
1800 func call(fn, arg unsafe.Pointer, n uint32)
1801 func ifaceE2I(t *runtimeType, src interface{}, dst unsafe.Pointer)
1802
1803 // Dummy annotation marking that the value x escapes,
1804 // for use in cases where the reflect code is so clever that
1805 // the compiler cannot follow.
1806 func escapes(x interface{}) {
1807     if dummy.b {
1808         dummy.x = x
1809     }
1810 }
1811
1812 var dummy struct {
1813     b bool
1814     x interface{}
1815 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/regexp/exec.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package regexp
6
7 import (
8     "io"
9     "regexp/syntax"
10 )
11
12 // A queue is a 'sparse array' holding pending threads of ex
13 // See http://research.swtch.com/2008/03/using-uninitialized
14 type queue struct {
15     sparse []uint32
16     dense []entry
17 }
18
19 // A entry is an entry on a queue.
20 // It holds both the instruction pc and the actual thread.
21 // Some queue entries are just place holders so that the mac
22 // knows it has considered that pc. Such entries have t ==
23 type entry struct {
24     pc uint32
25     t *thread
26 }
27
28 // A thread is the state of a single path through the machin
29 // an instruction and a corresponding capture array.
30 // See http://swtch.com/~rsc/regexp/regexp2.html
31 type thread struct {
32     inst *syntax.Inst
33     cap []int
34 }
35
36 // A machine holds all the state during an NFA simulation fo
37 type machine struct {
38     re *Regexp // corresponding Regexp
39     p *syntax.Prog // compiled program
40     q0, q1 queue // two queues for runq, nextq
41     pool []*thread // pool of available threads
42     matched bool // whether a match was found
43     matchcap []int // capture information for the
44
```

```

45         // cached inputs, to avoid allocation
46         inputBytes  inputBytes
47         inputString inputString
48         inputReader inputReader
49     }
50
51     func (m *machine) newInputBytes(b []byte) input {
52         m.inputBytes.str = b
53         return &m.inputBytes
54     }
55
56     func (m *machine) newInputString(s string) input {
57         m.inputString.str = s
58         return &m.inputString
59     }
60
61     func (m *machine) newInputReader(r io.RuneReader) input {
62         m.inputReader.r = r
63         m.inputReader.atEOT = false
64         m.inputReader.pos = 0
65         return &m.inputReader
66     }
67
68     // progMachine returns a new machine running the prog p.
69     func progMachine(p *syntax.Prog) *machine {
70         m := &machine{p: p}
71         n := len(m.p.Inst)
72         m.q0 = queue{make([]uint32, n), make([]entry, 0, n)}
73         m.q1 = queue{make([]uint32, n), make([]entry, 0, n)}
74         ncap := p.NumCap
75         if ncap < 2 {
76             ncap = 2
77         }
78         m.matchcap = make([]int, ncap)
79         return m
80     }
81
82     func (m *machine) init(ncap int) {
83         for _, t := range m.pool {
84             t.cap = t.cap[:ncap]
85         }
86         m.matchcap = m.matchcap[:ncap]
87     }
88
89     // alloc allocates a new thread with the given instruction.
90     // It uses the free pool if possible.
91     func (m *machine) alloc(i *syntax.Inst) *thread {
92         var t *thread
93         if n := len(m.pool); n > 0 {
94             t = m.pool[n-1]

```

```

95         m.pool = m.pool[:n-1]
96     } else {
97         t = new(thread)
98         t.cap = make([]int, len(m.matchcap), cap(m.m
99     })
100    t.inst = i
101    return t
102 }
103
104 // free returns t to the free pool.
105 func (m *machine) free(t *thread) {
106     m.inputBytes.str = nil
107     m.inputString.str = ""
108     m.inputReader.r = nil
109     m.pool = append(m.pool, t)
110 }
111
112 // match runs the machine over the input starting at pos.
113 // It reports whether a match was found.
114 // If so, m.matchcap holds the submatch information.
115 func (m *machine) match(i input, pos int) bool {
116     startCond := m.re.cond
117     if startCond == ^syntax.EmptyOp(0) { // impossible
118         return false
119     }
120     m.matched = false
121     for i := range m.matchcap {
122         m.matchcap[i] = -1
123     }
124     runq, nextq := &m.q0, &m.q1
125     r, r1 := endOfText, endOfText
126     width, width1 := 0, 0
127     r, width = i.step(pos)
128     if r != endOfText {
129         r1, width1 = i.step(pos + width)
130     }
131     var flag syntax.EmptyOp
132     if pos == 0 {
133         flag = syntax.EmptyOpContext(-1, r)
134     } else {
135         flag = i.context(pos)
136     }
137     for {
138         if len(runq.dense) == 0 {
139             if startCond&syntax.EmptyBeginText !
140                 // Anchored match, past begi
141                 break
142             }
143         if m.matched {

```

```

144             // Have match; finished expl
145             break
146         }
147         if len(m.re.prefix) > 0 && r1 != m.r
148             // Match requires literal pr
149             advance := i.index(m.re, pos
150             if advance < 0 {
151                 break
152             }
153             pos += advance
154             r, width = i.step(pos)
155             r1, width1 = i.step(pos + wi
156         }
157     }
158     if !m.matched {
159         if len(m.matchcap) > 0 {
160             m.matchcap[0] = pos
161         }
162         m.add(runq, uint32(m.p.Start), pos,
163     }
164     flag = syntax.EmptyOpContext(r, r1)
165     m.step(runq, nextq, pos, pos+width, r, flag)
166     if width == 0 {
167         break
168     }
169     if len(m.matchcap) == 0 && m.matched {
170         // Found a match and not paying atte
171         // to where it is, so any match will
172         break
173     }
174     pos += width
175     r, width = r1, width1
176     if r != endOfText {
177         r1, width1 = i.step(pos + width)
178     }
179     runq, nextq = nextq, runq
180 }
181 m.clear(nextq)
182 return m.matched
183 }
184
185 // clear frees all threads on the thread queue.
186 func (m *machine) clear(q *queue) {
187     for _, d := range q.dense {
188         if d.t != nil {
189             // m.free(d.t)
190             m.pool = append(m.pool, d.t)
191         }
192     }

```

```

193         q.dense = q.dense[:0]
194     }
195
196     // step executes one step of the machine, running each of th
197     // on runq and appending new threads to nextq.
198     // The step processes the rune c (which may be endOfText),
199     // which starts at position pos and ends at nextPos.
200     // nextCond gives the setting for the empty-width flags afte
201     func (m *machine) step(runq, nextq *queue, pos, nextPos int,
202         longest := m.re.longest
203         for j := 0; j < len(runq.dense); j++ {
204             d := &runq.dense[j]
205             t := d.t
206             if t == nil {
207                 continue
208             }
209             if longest && m.matched && len(t.cap) > 0 &&
210                 // m.free(t)
211                 m.pool = append(m.pool, t)
212                 continue
213             }
214             i := t.inst
215             add := false
216             switch i.Op {
217             default:
218                 panic("bad inst")
219
220             case syntax.InstMatch:
221                 if len(t.cap) > 0 && (!longest || !m
222                     t.cap[1] = pos
223                     copy(m.matchcap, t.cap)
224                 }
225                 if !longest {
226                     // First-match mode: cut off
227                     for _, d := range runq.dense
228                         if d.t != nil {
229                             // m.free(d.
230                                 m.pool = app
231                             }
232                         }
233                     runq.dense = runq.dense[:0]
234                 }
235                 m.matched = true
236
237             case syntax.InstRune:
238                 add = i.MatchRune(c)
239             case syntax.InstRune1:
240                 add = c == i.Rune[0]
241             case syntax.InstRuneAny:
242                 add = true

```

```

243         case syntax.InstRuneAnyNotNL:
244             add = c != '\n'
245         }
246         if add {
247             t = m.add(nextq, i.Out, nextPos, t.c
248         }
249         if t != nil {
250             // m.free(t)
251             m.pool = append(m.pool, t)
252         }
253     }
254     runq.dense = runq.dense[:0]
255 }
256
257 // add adds an entry to q for pc, unless the q already has s
258 // It also recursively adds an entry for all instructions re
259 // empty-width conditions satisfied by cond. pos gives the
260 // in the input.
261 func (m *machine) add(q *queue, pc uint32, pos int, cap []in
262     if pc == 0 {
263         return t
264     }
265     if j := q.sparse[pc]; j < uint32(len(q.dense)) && q.
266         return t
267     }
268
269     j := len(q.dense)
270     q.dense = q.dense[:j+1]
271     d := &q.dense[j]
272     d.t = nil
273     d.pc = pc
274     q.sparse[pc] = uint32(j)
275
276     i := &m.p.Inst[pc]
277     switch i.Op {
278     default:
279         panic("unhandled")
280     case syntax.InstFail:
281         // nothing
282     case syntax.InstAlt, syntax.InstAltMatch:
283         t = m.add(q, i.Out, pos, cap, cond, t)
284         t = m.add(q, i.Arg, pos, cap, cond, t)
285     case syntax.InstEmptyWidth:
286         if syntax.EmptyOp(i.Arg)&^cond == 0 {
287             t = m.add(q, i.Out, pos, cap, cond,
288         }
289     case syntax.InstNop:
290         t = m.add(q, i.Out, pos, cap, cond, t)
291     case syntax.InstCapture:

```

```

292         if int(i.Arg) < len(cap) {
293             opos := cap[i.Arg]
294             cap[i.Arg] = pos
295             m.add(q, i.Out, pos, cap, cond, nil)
296             cap[i.Arg] = opos
297         } else {
298             t = m.add(q, i.Out, pos, cap, cond,
299             }
300     case syntax.InstMatch, syntax.InstRune, syntax.InstR
301         if t == nil {
302             t = m.alloc(i)
303         } else {
304             t.inst = i
305         }
306         if len(cap) > 0 && &t.cap[0] != &cap[0] {
307             copy(t.cap, cap)
308         }
309         d.t = t
310         t = nil
311     }
312     return t
313 }
314
315 // empty is a non-nil 0-element slice,
316 // so doExecute can avoid an allocation
317 // when 0 captures are requested from a successful match.
318 var empty = make([]int, 0)
319
320 // doExecute finds the leftmost match in the input and retur
321 // the position of its subexpressions.
322 func (re *Regexp) doExecute(r io.RuneReader, b []byte, s str
323     m := re.get()
324     var i input
325     if r != nil {
326         i = m.newInputReader(r)
327     } else if b != nil {
328         i = m.newInputBytes(b)
329     } else {
330         i = m.newInputString(s)
331     }
332     m.init(ncap)
333     if !m.match(i, pos) {
334         re.put(m)
335         return nil
336     }
337     if ncap == 0 {
338         re.put(m)
339         return empty // empty but not nil
340     }

```

```
341         cap := make([]int, ncap)
342         copy(cap, m.matchcap)
343         re.put(m)
344         return cap
345     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/regexp/regexp.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package regexp implements regular expression search.
6 //
7 // The syntax of the regular expressions accepted is the same
8 // general syntax used by Perl, Python, and other languages.
9 // More precisely, it is the syntax accepted by RE2 and described
10 // at http://code.google.com/p/re2/wiki/Syntax, except for \C.
11 //
12 // All characters are UTF-8-encoded code points.
13 //
14 // There are 16 methods of Regexp that match a regular expression
15 // and return the matched text. Their names are matched by this regular
16 // expression:
17 //     Find(All)?(String)?(Submatch)?(Index)?
18 //
19 // If 'All' is present, the routine matches successive non-overlapping
20 // matches of the entire expression. Empty matches (whenever the
21 // expression is empty) are ignored. The return value is a slice containing
22 // the return values of the corresponding non-'All' routine. The
23 // routine takes an extra integer argument, n; if n >= 0, the function returns
24 // at most n matches/submatches.
25 //
26 // If 'String' is present, the argument is a string; otherwise, it is a
27 // slice of bytes; return values are adjusted as appropriate.
28 //
29 // If 'Submatch' is present, the return value is a slice identifying
30 // successive submatches of the expression. Submatches are identified by
31 // parenthesized subexpressions within the regular expression. The
32 // slice is ordered left to right in order of opening parenthesis. Submatch
33 // 0 is the entire expression, submatch 1 the match of the first
34 // parenthesized subexpression, and so on.
35 //
36 // If 'Index' is present, matches and submatches are identified by
37 // byte index pairs within the input string: result[2*n:2*n+1] identifies
38 // the nth submatch. The pair for n==0 identifies the match of the
39 // entire expression. If 'Index' is not present, the match is identified
40 // by the text of the match/submatch. If an index is negative, it
41 // indicates that the subexpression did not match any string in the input.
42 //
43 // There is also a subset of the methods that can be applied to a
44 // RuneReader:
```

```

45 //
46 //      MatchReader, FindReaderIndex, FindReaderSubmatchInde
47 //
48 // This set may grow. Note that regular expression matches
49 // examine text beyond the text returned by a match, so the
50 // match text from a RuneReader may read arbitrarily far int
51 // before returning.
52 //
53 // (There are a few other methods that do not match this pat
54 //
55 package regexp
56
57 import (
58     "bytes"
59     "io"
60     "regexp/syntax"
61     "strconv"
62     "strings"
63     "sync"
64     "unicode"
65     "unicode/utf8"
66 )
67
68 var debug = false
69
70 // Regexp is the representation of a compiled regular expres
71 // The public interface is entirely through methods.
72 // A Regexp is safe for concurrent use by multiple goroutine
73 type Regexp struct {
74     // read-only after Compile
75     expr      string      // as passed to Compile
76     prog      *syntax.Prog // compiled program
77     prefix    string      // required prefix in
78     prefixBytes []byte     // prefix, as a []byte
79     prefixComplete bool       // prefix is the entire
80     prefixRune rune       // first rune in prefix
81     cond      syntax.EmptyOp // empty-width condition
82     numSubexp int
83     subexpNames []string
84     longest    bool
85
86     // cache of machines for running regexp
87     mu      sync.Mutex
88     machine []*machine
89 }
90
91 // String returns the source text used to compile the regular
92 func (re *Regexp) String() string {
93     return re.expr
94 }

```

```

95
96 // Compile parses a regular expression and returns, if succe
97 // a Regexp object that can be used to match against text.
98 //
99 // When matching against text, the regexp returns a match th
100 // begins as early as possible in the input (leftmost), and
101 // it chooses the one that a backtracking search would have
102 // This so-called leftmost-first matching is the same semant
103 // that Perl, Python, and other implementations use, althoug
104 // package implements it without the expense of backtracking
105 // For POSIX leftmost-longest matching, see CompilePOSIX.
106 func Compile(expr string) (*Regexp, error) {
107     return compile(expr, syntax.Perl, false)
108 }
109
110 // CompilePOSIX is like Compile but restricts the regular ex
111 // to POSIX ERE (egrep) syntax and changes the match semanti
112 // leftmost-longest.
113 //
114 // That is, when matching against text, the regexp returns a
115 // begins as early as possible in the input (leftmost), and
116 // it chooses a match that is as long as possible.
117 // This so-called leftmost-longest matching is the same sema
118 // that early regular expression implementations used and th
119 // specifies.
120 //
121 // However, there can be multiple leftmost-longest matches,
122 // submatch choices, and here this package diverges from POS
123 // Among the possible leftmost-longest matches, this package
124 // the one that a backtracking search would have found first
125 // specifies that the match be chosen to maximize the length
126 // subexpression, then the second, and so on from left to ri
127 // The POSIX rule is computationally prohibitive and not eve
128 // See http://swtch.com/~rsc/regexp/regexp2.html#posix for d
129 func CompilePOSIX(expr string) (*Regexp, error) {
130     return compile(expr, syntax.POSIX, true)
131 }
132
133 func compile(expr string, mode syntax.Flags, longest bool) (
134     re, err := syntax.Parse(expr, mode)
135     if err != nil {
136         return nil, err
137     }
138     maxCap := re.MaxCap()
139     capNames := re.CapNames()
140
141     re = re.Simplify()
142     prog, err := syntax.Compile(re)
143     if err != nil {

```

```

144         return nil, err
145     }
146     regexp := &Regexp{
147         expr:      expr,
148         prog:      prog,
149         numSubexp: maxCap,
150         subexpNames: capNames,
151         cond:      prog.StartCond(),
152         longest:   longest,
153     }
154     regexp.prefix, regexp.prefixComplete = prog.Prefix()
155     if regexp.prefix != "" {
156         // TODO(rsc): Remove this allocation by addi
157         // IndexString to package bytes.
158         regexp.prefixBytes = []byte(regexp.prefix)
159         regexp.prefixRune, _ = utf8.DecodeRuneInStri
160     }
161     return regexp, nil
162 }
163
164 // get returns a machine to use for matching re.
165 // It uses the re's machine cache if possible, to avoid
166 // unnecessary allocation.
167 func (re *Regexp) get() *machine {
168     re.mu.Lock()
169     if n := len(re.machine); n > 0 {
170         z := re.machine[n-1]
171         re.machine = re.machine[:n-1]
172         re.mu.Unlock()
173         return z
174     }
175     re.mu.Unlock()
176     z := progMachine(re.prog)
177     z.re = re
178     return z
179 }
180
181 // put returns a machine to the re's machine cache.
182 // There is no attempt to limit the size of the cache, so it
183 // grow to the maximum number of simultaneous matches
184 // run using re. (The cache empties when re gets garbage co
185 func (re *Regexp) put(z *machine) {
186     re.mu.Lock()
187     re.machine = append(re.machine, z)
188     re.mu.Unlock()
189 }
190
191 // MustCompile is like Compile but panics if the expression
192 // It simplifies safe initialization of global variables hol

```

```

193 // expressions.
194 func MustCompile(str string) *Regexp {
195     regexp, error := Compile(str)
196     if error != nil {
197         panic(`regexp: Compile(` + quote(str) + `):
198     }
199     return regexp
200 }
201
202 // MustCompilePOSIX is like CompilePOSIX but panics if the e
203 // It simplifies safe initialization of global variables hol
204 // expressions.
205 func MustCompilePOSIX(str string) *Regexp {
206     regexp, error := CompilePOSIX(str)
207     if error != nil {
208         panic(`regexp: CompilePOSIX(` + quote(str) +
209     }
210     return regexp
211 }
212
213 func quote(s string) string {
214     if strconv.CanBackquote(s) {
215         return "`" + s + "`"
216     }
217     return strconv.Quote(s)
218 }
219
220 // NumSubexp returns the number of parenthesized subexpressi
221 func (re *Regexp) NumSubexp() int {
222     return re.numSubexp
223 }
224
225 // SubexpNames returns the names of the parenthesized subexp
226 // in this Regexp. The name for the first sub-expression is
227 // so that if m is a match slice, the name for m[i] is Subex
228 // Since the Regexp as a whole cannot be named, names[0] is
229 // the empty string. The slice should not be modified.
230 func (re *Regexp) SubexpNames() []string {
231     return re.subexpNames
232 }
233
234 const endOfText rune = -1
235
236 // input abstracts different representations of the input te
237 // one-character lookahead.
238 type input interface {
239     step(pos int) (r rune, width int) // advance one run
240     canCheckPrefix() bool             // can we look ahe
241     hasPrefix(re *Regexp) bool
242     index(re *Regexp, pos int) int

```

```

243         context(pos int) syntax.EmptyOp
244     }
245
246 // inputString scans a string.
247 type inputString struct {
248     str string
249 }
250
251 func (i *inputString) step(pos int) (rune, int) {
252     if pos < len(i.str) {
253         c := i.str[pos]
254         if c < utf8.RuneSelf {
255             return rune(c), 1
256         }
257         return utf8.DecodeRuneInString(i.str[pos:])
258     }
259     return endOfText, 0
260 }
261
262 func (i *inputString) canCheckPrefix() bool {
263     return true
264 }
265
266 func (i *inputString) hasPrefix(re *Regexp) bool {
267     return strings.HasPrefix(i.str, re.prefix)
268 }
269
270 func (i *inputString) index(re *Regexp, pos int) int {
271     return strings.Index(i.str[pos:], re.prefix)
272 }
273
274 func (i *inputString) context(pos int) syntax.EmptyOp {
275     r1, r2 := endOfText, endOfText
276     if pos > 0 && pos <= len(i.str) {
277         r1, _ = utf8.DecodeLastRuneInString(i.str[:pos])
278     }
279     if pos < len(i.str) {
280         r2, _ = utf8.DecodeRuneInString(i.str[pos:])
281     }
282     return syntax.EmptyOpContext(r1, r2)
283 }
284
285 // inputBytes scans a byte slice.
286 type inputBytes struct {
287     str []byte
288 }
289
290 func (i *inputBytes) step(pos int) (rune, int) {
291     if pos < len(i.str) {

```

```

292         c := i.str[pos]
293         if c < utf8.RuneSelf {
294             return rune(c), 1
295         }
296         return utf8.DecodeRune(i.str[pos:])
297     }
298     return endOfText, 0
299 }
300
301 func (i *inputBytes) canCheckPrefix() bool {
302     return true
303 }
304
305 func (i *inputBytes) hasPrefix(re *Regexp) bool {
306     return bytes.HasPrefix(i.str, re.prefixBytes)
307 }
308
309 func (i *inputBytes) index(re *Regexp, pos int) int {
310     return bytes.Index(i.str[pos:], re.prefixBytes)
311 }
312
313 func (i *inputBytes) context(pos int) syntax.EmptyOp {
314     r1, r2 := endOfText, endOfText
315     if pos > 0 && pos <= len(i.str) {
316         r1, _ = utf8.DecodeLastRune(i.str[:pos])
317     }
318     if pos < len(i.str) {
319         r2, _ = utf8.DecodeRune(i.str[pos:])
320     }
321     return syntax.EmptyOpContext(r1, r2)
322 }
323
324 // inputReader scans a RuneReader.
325 type inputReader struct {
326     r      io.RuneReader
327     atEOT bool
328     pos   int
329 }
330
331 func (i *inputReader) step(pos int) (rune, int) {
332     if !i.atEOT && pos != i.pos {
333         return endOfText, 0
334     }
335 }
336 r, w, err := i.r.ReadRune()
337 if err != nil {
338     i.atEOT = true
339     return endOfText, 0
340 }

```

```

341         i.pos += w
342         return r, w
343     }
344
345     func (i *inputReader) canCheckPrefix() bool {
346         return false
347     }
348
349     func (i *inputReader) hasPrefix(re *Regexp) bool {
350         return false
351     }
352
353     func (i *inputReader) index(re *Regexp, pos int) int {
354         return -1
355     }
356
357     func (i *inputReader) context(pos int) syntax.EmptyOp {
358         return 0
359     }
360
361     // LiteralPrefix returns a literal string that must begin an
362     // of the regular expression re. It returns the boolean true
363     // if the literal string comprises the entire regular expression.
364     func (re *Regexp) LiteralPrefix() (prefix string, complete bool) {
365         return re.prefix, re.prefixComplete
366     }
367
368     // MatchReader returns whether the Regexp matches the text r
369     // from a RuneReader. The return value is a boolean: true for match
370     // and false otherwise.
371     func (re *Regexp) MatchReader(r io.RuneReader) bool {
372         return re.doExecute(r, nil, "", 0, 0) != nil
373     }
374
375     // MatchString returns whether the Regexp matches the string s.
376     // The return value is a boolean: true for match, false for
377     // no match.
378     func (re *Regexp) MatchString(s string) bool {
379         return re.doExecute(nil, nil, s, 0, 0) != nil
380     }
381
382     // Match returns whether the Regexp matches the byte slice b.
383     // The return value is a boolean: true for match, false for
384     // no match.
385     func (re *Regexp) Match(b []byte) bool {
386         return re.doExecute(nil, b, "", 0, 0) != nil
387     }
388
389     // MatchReader checks whether a textual regular expression re
390     // is matched by the RuneReader. More complicated queries need to
391     // use the full Regexp interface.
392     func MatchReader(pattern string, r io.RuneReader) (matched bool,

```

```

391         re, err := Compile(pattern)
392         if err != nil {
393             return false, err
394         }
395         return re.MatchReader(r), nil
396     }
397
398     // MatchString checks whether a textual regular expression
399     // matches a string. More complicated queries need
400     // to use Compile and the full Regexp interface.
401     func MatchString(pattern string, s string) (matched bool, er
402         re, err := Compile(pattern)
403         if err != nil {
404             return false, err
405         }
406         return re.MatchString(s), nil
407     }
408
409     // Match checks whether a textual regular expression
410     // matches a byte slice. More complicated queries need
411     // to use Compile and the full Regexp interface.
412     func Match(pattern string, b []byte) (matched bool, error er
413         re, err := Compile(pattern)
414         if err != nil {
415             return false, err
416         }
417         return re.Match(b), nil
418     }
419
420     // ReplaceAllString returns a copy of src, replacing matches
421     // with the replacement string repl. Inside repl, $ signs a
422     // in Expand, so for instance $1 represents the text of the
423     func (re *Regexp) ReplaceAllString(src, repl string) string
424         n := 2
425         if strings.Index(repl, "$") >= 0 {
426             n = 2 * (re.numSubexp + 1)
427         }
428         b := re.replaceAll(nil, src, n, func(dst []byte, mat
429             return re.expand(dst, repl, nil, src, match)
430         })
431         return string(b)
432     }
433
434     // ReplaceAllStringLiteral returns a copy of src, replacing
435     // with the replacement string repl. The replacement repl i
436     // without using Expand.
437     func (re *Regexp) ReplaceAllLiteralString(src, repl string)
438         return string(re.replaceAll(nil, src, 2, func(dst []
439             return append(dst, repl...)

```

```

440         )))
441     }
442
443     // ReplaceAllStringFunc returns a copy of src in which all m
444     // Regexp have been replaced by the return value of of funct
445     // to the matched substring. The replacement returned by re
446     // directly, without using Expand.
447     func (re *Regexp) ReplaceAllStringFunc(src string, repl func
448         b := re.replaceAll(nil, src, 2, func(dst []byte, mat
449             return append(dst, repl(src[match[0]:match[1]
450         })
451     return string(b)
452 }
453
454 func (re *Regexp) replaceAll(bsrc []byte, src string, nmatch
455     lastMatchEnd := 0 // end position of the most recent
456     searchPos := 0    // position where we next look for
457     var buf []byte
458     var endPos int
459     if bsrc != nil {
460         endPos = len(bsrc)
461     } else {
462         endPos = len(src)
463     }
464     for searchPos <= endPos {
465         a := re.doExecute(nil, bsrc, src, searchPos,
466             if len(a) == 0 {
467                 break // no more matches
468             }
469
470             // Copy the unmatched characters before this
471             if bsrc != nil {
472                 buf = append(buf, bsrc[lastMatchEnd:
473             } else {
474                 buf = append(buf, src[lastMatchEnd:a
475             }
476
477             // Now insert a copy of the replacement stri
478             // match of the empty string immediately aft
479             // (Otherwise, we get double replacement for
480             // match both empty and nonempty strings.)
481             if a[1] > lastMatchEnd || a[0] == 0 {
482                 buf = repl(buf, a)
483             }
484             lastMatchEnd = a[1]
485
486             // Advance past this match; always advance a
487             var width int
488             if bsrc != nil {

```

```

489         _, width = utf8.DecodeRune(bsrc[sear
490     } else {
491         _, width = utf8.DecodeRuneInString(s
492     }
493     if searchPos+width > a[1] {
494         searchPos += width
495     } else if searchPos+1 > a[1] {
496         // This clause is only needed at the
497         // string. In that case, DecodeRune
498         searchPos++
499     } else {
500         searchPos = a[1]
501     }
502 }
503
504 // Copy the unmatched characters after the last matc
505 if bsrc != nil {
506     buf = append(buf, bsrc[lastMatchEnd:]...)
507 } else {
508     buf = append(buf, src[lastMatchEnd:]...)
509 }
510
511 return buf
512 }
513
514 // ReplaceAll returns a copy of src, replacing matches of th
515 // with the replacement string repl. Inside repl, $ signs a
516 // in Expand, so for instance $1 represents the text of the
517 func (re *Regexp) ReplaceAll(src, repl []byte) []byte {
518     n := 2
519     if bytes.IndexByte(repl, '$') >= 0 {
520         n = 2 * (re.numSubexp + 1)
521     }
522     srepl := ""
523     b := re.replaceAll(src, "", n, func(dst []byte, matc
524         if len(srepl) != len(repl) {
525             srepl = string(repl)
526         }
527         return re.expand(dst, srepl, src, "", match)
528     })
529     return b
530 }
531
532 // ReplaceAllLiteral returns a copy of src, replacing matche
533 // with the replacement bytes repl. The replacement repl is
534 // without using Expand.
535 func (re *Regexp) ReplaceAllLiteral(src, repl []byte) []byte
536     return re.replaceAll(src, "", 2, func(dst []byte, ma
537         return append(dst, repl...)
538     })

```

```

539 }
540
541 // ReplaceAllFunc returns a copy of src in which all matches
542 // Regexp have been replaced by the return value of of funct
543 // to the matched byte slice. The replacement returned by r
544 // directly, without using Expand.
545 func (re *Regexp) ReplaceAllFunc(src []byte, repl func([]byte)
546     return re.replaceAll(src, "", 2, func(dst []byte, ma
547         return append(dst, repl(src[match[0]:match[1
548     })
549 }
550
551 var specialBytes = []byte(`\.*?()|[\{\}^\$`)
552
553 func special(b byte) bool {
554     return bytes.IndexByte(specialBytes, b) >= 0
555 }
556
557 // QuoteMeta returns a string that quotes all regular expres
558 // inside the argument text; the returned string is a regula
559 // the literal text. For example, QuoteMeta(`[foo]`) return
560 func QuoteMeta(s string) string {
561     b := make([]byte, 2*len(s))
562
563     // A byte loop is correct because all metacharacters
564     j := 0
565     for i := 0; i < len(s); i++ {
566         if special(s[i]) {
567             b[j] = '\\'
568             j++
569         }
570         b[j] = s[i]
571         j++
572     }
573     return string(b[0:j])
574 }
575
576 // The number of capture values in the program may correspon
577 // to fewer capturing expressions than are in the regexp.
578 // For example, "(a){0}" turns into an empty program, so the
579 // maximum capture in the program is 0 but we need to return
580 // an expression for \1. Pad appends -1s to the slice a as
581 func (re *Regexp) pad(a []int) []int {
582     if a == nil {
583         // No match.
584         return nil
585     }
586     n := (1 + re.numSubexp) * 2
587     for len(a) < n {

```

```

588             a = append(a, -1)
589         }
590     return a
591 }
592
593 // Find matches in slice b if b is non-nil, otherwise find m
594 func (re *Regexp) allMatches(s string, b []byte, n int, deli
595     var end int
596     if b == nil {
597         end = len(s)
598     } else {
599         end = len(b)
600     }
601
602     for pos, i, prevMatchEnd := 0, 0, -1; i < n && pos <
603         matches := re.doExecute(nil, b, s, pos, re.p
604         if len(matches) == 0 {
605             break
606         }
607
608         accept := true
609         if matches[1] == pos {
610             // We've found an empty match.
611             if matches[0] == prevMatchEnd {
612                 // We don't allow an empty m
613                 // after a previous match, s
614                 accept = false
615             }
616             var width int
617             // TODO: use step()
618             if b == nil {
619                 _, width = utf8.DecodeRuneIn
620             } else {
621                 _, width = utf8.DecodeRune(b
622             }
623             if width > 0 {
624                 pos += width
625             } else {
626                 pos = end + 1
627             }
628         } else {
629             pos = matches[1]
630         }
631         prevMatchEnd = matches[1]
632
633         if accept {
634             deliver(re.pad(matches))
635             i++
636         }

```

```

637     }
638 }
639
640 // Find returns a slice holding the text of the leftmost mat
641 // A return value of nil indicates no match.
642 func (re *Regexp) Find(b []byte) []byte {
643     a := re.doExecute(nil, b, "", 0, 2)
644     if a == nil {
645         return nil
646     }
647     return b[a[0]:a[1]]
648 }
649
650 // FindIndex returns a two-element slice of integers definin
651 // the leftmost match in b of the regular expression. The m
652 // b[loc[0]:loc[1]].
653 // A return value of nil indicates no match.
654 func (re *Regexp) FindIndex(b []byte) (loc []int) {
655     a := re.doExecute(nil, b, "", 0, 2)
656     if a == nil {
657         return nil
658     }
659     return a[0:2]
660 }
661
662 // FindString returns a string holding the text of the leftm
663 // expression. If there is no match, the return value is an
664 // but it will also be empty if the regular expression succe
665 // an empty string. Use FindStringIndex or FindStringSubmat
666 // necessary to distinguish these cases.
667 func (re *Regexp) FindString(s string) string {
668     a := re.doExecute(nil, nil, s, 0, 2)
669     if a == nil {
670         return ""
671     }
672     return s[a[0]:a[1]]
673 }
674
675 // FindStringIndex returns a two-element slice of integers d
676 // location of the leftmost match in s of the regular expres
677 // itself is at s[loc[0]:loc[1]].
678 // A return value of nil indicates no match.
679 func (re *Regexp) FindStringIndex(s string) (loc []int) {
680     a := re.doExecute(nil, nil, s, 0, 2)
681     if a == nil {
682         return nil
683     }
684     return a[0:2]
685 }
686

```

```

687 // FindReaderIndex returns a two-element slice of integers d
688 // location of the leftmost match of the regular expression
689 // the RuneReader. The match itself is at s[loc[0]:loc[1]].
690 // value of nil indicates no match.
691 func (re *Regexp) FindReaderIndex(r io.RuneReader) (loc []int) {
692     a := re.doExecute(r, nil, "", 0, 2)
693     if a == nil {
694         return nil
695     }
696     return a[0:2]
697 }
698
699 // FindSubmatch returns a slice of slices holding the text o
700 // match of the regular expression in b and the matches, if
701 // subexpressions, as defined by the 'Submatch' descriptions
702 // comment.
703 // A return value of nil indicates no match.
704 func (re *Regexp) FindSubmatch(b []byte) [][]byte {
705     a := re.doExecute(nil, b, "", 0, re.prog.NumCap)
706     if a == nil {
707         return nil
708     }
709     ret := make([][]byte, 1+re.numSubexp)
710     for i := range ret {
711         if 2*i < len(a) && a[2*i] >= 0 {
712             ret[i] = b[a[2*i]:a[2*i+1]]
713         }
714     }
715     return ret
716 }
717
718 // Expand appends template to dst and returns the result; du
719 // append, Expand replaces variables in the template with co
720 // matches drawn from src. The match slice should have been
721 // FindSubmatchIndex.
722 //
723 // In the template, a variable is denoted by a substring of
724 // $name or ${name}, where name is a non-empty sequence of 1
725 // digits, and underscores. A purely numeric name like $1 r
726 // the submatch with the corresponding index; other names re
727 // capturing parentheses named with the (?P<name>...) syntax
728 // reference to an out of range or unmatched index or a name
729 // present in the regular expression is replaced with an emp
730 //
731 // In the $name form, name is taken to be as long as possibl
732 // equivalent to ${1x}, not ${1}x, and, $10 is equivalent to
733 //
734 // To insert a literal $ in the output, use $$ in the templa
735 func (re *Regexp) Expand(dst []byte, template []byte, src []

```

```

736         return re.expand(dst, string(template), src, "", mat
737     }
738
739     // ExpandString is like Expand but the template and source a
740     // It appends to and returns a byte slice in order to give t
741     // code control over allocation.
742     func (re *Regexp) ExpandString(dst []byte, template string,
743         return re.expand(dst, template, nil, src, match)
744     }
745
746     func (re *Regexp) expand(dst []byte, template string, bsrc [
747         for len(template) > 0 {
748             i := strings.Index(template, "$")
749             if i < 0 {
750                 break
751             }
752             dst = append(dst, template[:i]...)
753             template = template[i:]
754             if len(template) > 1 && template[1] == '$' {
755                 // Treat $$ as $.
756                 dst = append(dst, '$')
757                 template = template[2:]
758                 continue
759             }
760             name, num, rest, ok := extract(template)
761             if !ok {
762                 // Malformed; treat $ as raw text.
763                 dst = append(dst, '$')
764                 template = template[1:]
765                 continue
766             }
767             template = rest
768             if num >= 0 {
769                 if 2*num+1 < len(match) {
770                     if bsrc != nil {
771                         dst = append(dst, bs
772                     } else {
773                         dst = append(dst, sr
774                     }
775                 }
776             } else {
777                 for i, namei := range re.subexpNames
778                     if name == namei && 2*i+1 <
779                         if bsrc != nil {
780                             dst = append
781                         } else {
782                             dst = append
783                         }
784                 break

```

```

785         }
786     }
787 }
788 }
789     dst = append(dst, template...)
790     return dst
791 }
792
793 // extract returns the name from a leading "$name" or "${name"
794 // If it is a number, extract returns num set to that number
795 func extract(str string) (name string, num int, rest string,
796     if len(str) < 2 || str[0] != '$' {
797     return
798     }
799     brace := false
800     if str[1] == '{' {
801         brace = true
802         str = str[2:]
803     } else {
804         str = str[1:]
805     }
806     i := 0
807     for i < len(str) {
808         rune, size := utf8.DecodeRuneInString(str[i:]
809         if !unicode.IsLetter(rune) && !unicode.IsDig
810             break
811         }
812         i += size
813     }
814     if i == 0 {
815         // empty name is not okay
816         return
817     }
818     name = str[:i]
819     if brace {
820         if i >= len(str) || str[i] != '}' {
821             // missing closing brace
822             return
823         }
824         i++
825     }
826
827     // Parse number.
828     num = 0
829     for i := 0; i < len(name); i++ {
830         if name[i] < '0' || '9' < name[i] || num >=
831             num = -1
832             break
833         }
834         num = num*10 + int(name[i]) - '0'

```

```

835     }
836     // Disallow leading zeros.
837     if name[0] == '0' && len(name) > 1 {
838         num = -1
839     }
840
841     rest = str[i:]
842     ok = true
843     return
844 }
845
846 // FindSubmatchIndex returns a slice holding the index pairs
847 // leftmost match of the regular expression in b and the mat
848 // its subexpressions, as defined by the 'Submatch' and 'Ind
849 // in the package comment.
850 // A return value of nil indicates no match.
851 func (re *Regexp) FindSubmatchIndex(b []byte) []int {
852     return re.pad(re.doExecute(nil, b, "", 0, re.prog.Nu
853 }
854
855 // FindStringSubmatch returns a slice of strings holding the
856 // leftmost match of the regular expression in s and the mat
857 // its subexpressions, as defined by the 'Submatch' descript
858 // package comment.
859 // A return value of nil indicates no match.
860 func (re *Regexp) FindStringSubmatch(s string) []string {
861     a := re.doExecute(nil, nil, s, 0, re.prog.NumCap)
862     if a == nil {
863         return nil
864     }
865     ret := make([]string, 1+re.numSubexp)
866     for i := range ret {
867         if 2*i < len(a) && a[2*i] >= 0 {
868             ret[i] = s[a[2*i]:a[2*i+1]]
869         }
870     }
871     return ret
872 }
873
874 // FindStringSubmatchIndex returns a slice holding the index
875 // identifying the leftmost match of the regular expression
876 // matches, if any, of its subexpressions, as defined by the
877 // 'Index' descriptions in the package comment.
878 // A return value of nil indicates no match.
879 func (re *Regexp) FindStringSubmatchIndex(s string) []int {
880     return re.pad(re.doExecute(nil, nil, s, 0, re.prog.N
881 }
882
883 // FindReaderSubmatchIndex returns a slice holding the index

```

```

884 // identifying the leftmost match of the regular expression
885 // the RuneReader, and the matches, if any, of its subexpres
886 // by the 'Submatch' and 'Index' descriptions in the package
887 // return value of nil indicates no match.
888 func (re *Regexp) FindReaderSubmatchIndex(r io.RuneReader) [
889     return re.pad(re.doExecute(r, nil, "", 0, re.prog.Nu
890 }
891
892 const startSize = 10 // The size at which to start a slice i
893
894 // FindAll is the 'All' version of Find; it returns a slice
895 // matches of the expression, as defined by the 'All' descri
896 // package comment.
897 // A return value of nil indicates no match.
898 func (re *Regexp) FindAll(b []byte, n int) [][]byte {
899     if n < 0 {
900         n = len(b) + 1
901     }
902     result := make([][]byte, 0, startSize)
903     re.allMatches("", b, n, func(match []int) {
904         result = append(result, b[match[0]:match[1]])
905     })
906     if len(result) == 0 {
907         return nil
908     }
909     return result
910 }
911
912 // FindAllIndex is the 'All' version of FindIndex; it return
913 // successive matches of the expression, as defined by the '
914 // in the package comment.
915 // A return value of nil indicates no match.
916 func (re *Regexp) FindAllIndex(b []byte, n int) [][]int {
917     if n < 0 {
918         n = len(b) + 1
919     }
920     result := make([][]int, 0, startSize)
921     re.allMatches("", b, n, func(match []int) {
922         result = append(result, match[0:2])
923     })
924     if len(result) == 0 {
925         return nil
926     }
927     return result
928 }
929
930 // FindAllString is the 'All' version of FindString; it retu
931 // successive matches of the expression, as defined by the '
932 // in the package comment.

```

```

933 // A return value of nil indicates no match.
934 func (re *Regexp) FindAllString(s string, n int) []string {
935     if n < 0 {
936         n = len(s) + 1
937     }
938     result := make([]string, 0, startSize)
939     re.allMatches(s, nil, n, func(match []int) {
940         result = append(result, s[match[0]:match[1]])
941     })
942     if len(result) == 0 {
943         return nil
944     }
945     return result
946 }
947
948 // FindAllStringIndex is the 'All' version of FindStringIndex
949 // slice of all successive matches of the expression, as defined
950 // description in the package comment.
951 // A return value of nil indicates no match.
952 func (re *Regexp) FindAllStringIndex(s string, n int) [][]int {
953     if n < 0 {
954         n = len(s) + 1
955     }
956     result := make([][]int, 0, startSize)
957     re.allMatches(s, nil, n, func(match []int) {
958         result = append(result, match[0:2])
959     })
960     if len(result) == 0 {
961         return nil
962     }
963     return result
964 }
965
966 // FindAllSubmatch is the 'All' version of FindSubmatch; it
967 // slice of all successive matches of the expression, as defined by
968 // description in the package comment.
969 // A return value of nil indicates no match.
970 func (re *Regexp) FindAllSubmatch(b []byte, n int) [][][]byte {
971     if n < 0 {
972         n = len(b) + 1
973     }
974     result := make([][][]byte, 0, startSize)
975     re.allMatches("", b, n, func(match []int) {
976         slice := make([][]byte, len(match)/2)
977         for j := range slice {
978             if match[2*j] >= 0 {
979                 slice[j] = b[match[2*j]:match[2*j+1]]
980             }
981         }
982         result = append(result, slice)

```

```

983         })
984         if len(result) == 0 {
985             return nil
986         }
987         return result
988     }
989
990     // FindAllSubmatchIndex is the 'All' version of FindSubmatch
991     // a slice of all successive matches of the expression, as d
992     // 'All' description in the package comment.
993     // A return value of nil indicates no match.
994     func (re *Regexp) FindAllSubmatchIndex(b []byte, n int) [][]
995         if n < 0 {
996             n = len(b) + 1
997         }
998         result := make([][]int, 0, startSize)
999         re.allMatches("", b, n, func(match []int) {
1000             result = append(result, match)
1001         })
1002         if len(result) == 0 {
1003             return nil
1004         }
1005         return result
1006     }
1007
1008     // FindAllStringSubmatch is the 'All' version of FindStrings
1009     // returns a slice of all successive matches of the expressi
1010     // the 'All' description in the package comment.
1011     // A return value of nil indicates no match.
1012     func (re *Regexp) FindAllStringSubmatch(s string, n int) [][]
1013         if n < 0 {
1014             n = len(s) + 1
1015         }
1016         result := make([][]string, 0, startSize)
1017         re.allMatches(s, nil, n, func(match []int) {
1018             slice := make([]string, len(match)/2)
1019             for j := range slice {
1020                 if match[2*j] >= 0 {
1021                     slice[j] = s[match[2*j]:matc
1022                 }
1023             }
1024             result = append(result, slice)
1025         })
1026         if len(result) == 0 {
1027             return nil
1028         }
1029         return result
1030     }
1031

```

```
1032 // FindAllStringSubmatchIndex is the 'All' version of
1033 // FindStringSubmatchIndex; it returns a slice of all succes
1034 // the expression, as defined by the 'All' description in th
1035 // comment.
1036 // A return value of nil indicates no match.
1037 func (re *Regexp) FindAllStringSubmatchIndex(s string, n int
1038     if n < 0 {
1039         n = len(s) + 1
1040     }
1041     result := make([][]int, 0, startSize)
1042     re.allMatches(s, nil, n, func(match []int) {
1043         result = append(result, match)
1044     })
1045     if len(result) == 0 {
1046         return nil
1047     }
1048     return result
1049 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/regexp/syntax/compile.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package syntax
6
7 import "unicode"
8
9 // A patchList is a list of instruction pointers that need t
10 // Because the pointers haven't been filled in yet, we can r
11 // to hold the list. It's kind of sleazy, but works well in
12 // See http://swtch.com/~rsc/regexp/regexp1.html for inspira
13 //
14 // These aren't really pointers: they're integers, so we can
15 // this way without using package unsafe. A value l denotes
16 // p.inst[l>>1].Out (l&1==0) or .Arg (l&1==1).
17 // l == 0 denotes the empty list, okay because we start ever
18 // with a fail instruction, so we'll never want to point at
19 type patchList uint32
20
21 func (l patchList) next(p *Prog) patchList {
22     i := &p.Inst[l>>1]
23     if l&1 == 0 {
24         return patchList(i.Out)
25     }
26     return patchList(i.Arg)
27 }
28
29 func (l patchList) patch(p *Prog, val uint32) {
30     for l != 0 {
31         i := &p.Inst[l>>1]
32         if l&1 == 0 {
33             l = patchList(i.Out)
34             i.Out = val
35         } else {
36             l = patchList(i.Arg)
37             i.Arg = val
38         }
39     }
40 }
41
```

```

42 func (l1 patchList) append(p *Prog, l2 patchList) patchList
43     if l1 == 0 {
44         return l2
45     }
46     if l2 == 0 {
47         return l1
48     }
49
50     last := l1
51     for {
52         next := last.next(p)
53         if next == 0 {
54             break
55         }
56         last = next
57     }
58
59     i := &p.Inst[last>>1]
60     if last&1 == 0 {
61         i.Out = uint32(l2)
62     } else {
63         i.Arg = uint32(l2)
64     }
65     return l1
66 }
67
68 // A frag represents a compiled program fragment.
69 type frag struct {
70     i    uint32    // index of first instruction
71     out patchList // where to record end instruction
72 }
73
74 type compiler struct {
75     p *Prog
76 }
77
78 // Compile compiles the regexp into a program to be executed
79 // The regexp should have been simplified already (returned
80 func Compile(re *Regexp) (*Prog, error) {
81     var c compiler
82     c.init()
83     f := c.compile(re)
84     f.out.patch(c.p, c.inst(InstMatch).i)
85     c.p.Start = int(f.i)
86     return c.p, nil
87 }
88
89 func (c *compiler) init() {
90     c.p = new(Prog)
91     c.p.NumCap = 2 // implicit ( and ) for whole match $

```

```

92         c.inst(InstFail)
93     }
94
95     var anyRuneNotNL = []rune{0, '\n' - 1, '\n' + 1, unicode.Max
96     var anyRune = []rune{0, unicode.MaxRune}
97
98     func (c *compiler) compile(re *Regexp) frag {
99         switch re.Op {
100        case OpNoMatch:
101            return c.fail()
102        case OpEmptyMatch:
103            return c.nop()
104        case OpLiteral:
105            if len(re.Rune) == 0 {
106                return c.nop()
107            }
108            var f frag
109            for j := range re.Rune {
110                f1 := c.rune(re.Rune[j:j+1], re.Flag
111                if j == 0 {
112                    f = f1
113                } else {
114                    f = c.cat(f, f1)
115                }
116            }
117            return f
118        case OpCharClass:
119            return c.rune(re.Rune, re.Flags)
120        case OpAnyCharNotNL:
121            return c.rune(anyRuneNotNL, 0)
122        case OpAnyChar:
123            return c.rune(anyRune, 0)
124        case OpBeginLine:
125            return c.empty(EmptyBeginLine)
126        case OpEndLine:
127            return c.empty(EmptyEndLine)
128        case OpBeginText:
129            return c.empty(EmptyBeginText)
130        case OpEndText:
131            return c.empty(EmptyEndText)
132        case OpWordBoundary:
133            return c.empty(EmptyWordBoundary)
134        case OpNoWordBoundary:
135            return c.empty(EmptyNoWordBoundary)
136        case OpCapture:
137            bra := c.cap(uint32(re.Cap << 1))
138            sub := c.compile(re.Sub[0])
139            ket := c.cap(uint32(re.Cap<<1 | 1))
140            return c.cat(c.cat(bra, sub), ket)

```

```

141     case OpStar:
142         return c.star(c.compile(re.Sub[0]), re.Flags
143     case OpPlus:
144         return c.plus(c.compile(re.Sub[0]), re.Flags
145     case OpQuest:
146         return c.quest(c.compile(re.Sub[0]), re.Flag
147     case OpConcat:
148         if len(re.Sub) == 0 {
149             return c.nop()
150         }
151         var f frag
152         for i, sub := range re.Sub {
153             if i == 0 {
154                 f = c.compile(sub)
155             } else {
156                 f = c.cat(f, c.compile(sub))
157             }
158         }
159         return f
160     case OpAlternate:
161         var f frag
162         for _, sub := range re.Sub {
163             f = c.alt(f, c.compile(sub))
164         }
165         return f
166     }
167     panic("regexp: unhandled case in compile")
168 }
169
170 func (c *compiler) inst(op InstOp) frag {
171     // TODO: impose length limit
172     f := frag{i: uint32(len(c.p.Inst))}
173     c.p.Inst = append(c.p.Inst, Inst{Op: op})
174     return f
175 }
176
177 func (c *compiler) nop() frag {
178     f := c.inst(InstNop)
179     f.out = patchList(f.i << 1)
180     return f
181 }
182
183 func (c *compiler) fail() frag {
184     return frag{}
185 }
186
187 func (c *compiler) cap(arg uint32) frag {
188     f := c.inst(InstCapture)
189     f.out = patchList(f.i << 1)

```

```

190         c.p.Inst[f.i].Arg = arg
191
192         if c.p.NumCap < int(arg)+1 {
193             c.p.NumCap = int(arg) + 1
194         }
195         return f
196     }
197
198     func (c *compiler) cat(f1, f2 frag) frag {
199         // concat of failure is failure
200         if f1.i == 0 || f2.i == 0 {
201             return frag{}
202         }
203
204         // TODO: elide nop
205
206         f1.out.patch(c.p, f2.i)
207         return frag{f1.i, f2.out}
208     }
209
210     func (c *compiler) alt(f1, f2 frag) frag {
211         // alt of failure is other
212         if f1.i == 0 {
213             return f2
214         }
215         if f2.i == 0 {
216             return f1
217         }
218
219         f := c.inst(InstAlt)
220         i := &c.p.Inst[f.i]
221         i.Out = f1.i
222         i.Arg = f2.i
223         f.out = f1.out.append(c.p, f2.out)
224         return f
225     }
226
227     func (c *compiler) quest(f1 frag, nongreedy bool) frag {
228         f := c.inst(InstAlt)
229         i := &c.p.Inst[f.i]
230         if nongreedy {
231             i.Arg = f1.i
232             f.out = patchList(f.i << 1)
233         } else {
234             i.Out = f1.i
235             f.out = patchList(f.i<<1 | 1)
236         }
237         f.out = f.out.append(c.p, f1.out)
238         return f
239     }

```

```

240
241 func (c *compiler) star(f1 frag, nongreedy bool) frag {
242     f := c.inst(InstAlt)
243     i := &c.p.Inst[f.i]
244     if nongreedy {
245         i.Arg = f1.i
246         f.out = patchList(f.i << 1)
247     } else {
248         i.Out = f1.i
249         f.out = patchList(f.i<<1 | 1)
250     }
251     f1.out.patch(c.p, f.i)
252     return f
253 }
254
255 func (c *compiler) plus(f1 frag, nongreedy bool) frag {
256     return frag{f1.i, c.star(f1, nongreedy).out}
257 }
258
259 func (c *compiler) empty(op EmptyOp) frag {
260     f := c.inst(InstEmptyWidth)
261     c.p.Inst[f.i].Arg = uint32(op)
262     f.out = patchList(f.i << 1)
263     return f
264 }
265
266 func (c *compiler) rune(r []rune, flags Flags) frag {
267     f := c.inst(InstRune)
268     i := &c.p.Inst[f.i]
269     i.Rune = r
270     flags &= FoldCase // only relevant flag is FoldCase
271     if len(r) != 1 || unicode.SimpleFold(r[0]) == r[0] {
272         // and sometimes not even that
273         flags &^= FoldCase
274     }
275     i.Arg = uint32(flags)
276     f.out = patchList(f.i << 1)
277
278     // Special cases for exec machine.
279     switch {
280     case flags&FoldCase == 0 && (len(r) == 1 || len(r) =
281         i.Op = InstRune1
282     case len(r) == 2 && r[0] == 0 && r[1] == unicode.Max
283         i.Op = InstRuneAny
284     case len(r) == 4 && r[0] == 0 && r[1] == '\n'-1 && r
285         i.Op = InstRuneAnyNotNL
286     }
287
288     return f

```

289 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/regexp/syntax/parse.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package syntax parses regular expressions into parse tree
6 // parse trees into programs. Most clients of regular expres
7 // the facilities of package regexp (such as Compile and Mat
8 // this package.
9 package syntax
10
11 import (
12     "sort"
13     "strings"
14     "unicode"
15     "unicode/utf8"
16 )
17
18 // An Error describes a failure to parse a regular expressio
19 // and gives the offending expression.
20 type Error struct {
21     Code ErrorCode
22     Expr string
23 }
24
25 func (e *Error) Error() string {
26     return "error parsing regexp: " + e.Code.String() +
27 }
28
29 // An ErrorCode describes a failure to parse a regular expre
30 type ErrorCode string
31
32 const (
33     // Unexpected error
34     ErrInternalError ErrorCode = "regexp/syntax: interna
35
36     // Parse errors
37     ErrInvalidCharClass      ErrorCode = "invalid charac
38     ErrInvalidCharRange     ErrorCode = "invalid charac
39     ErrInvalidEscape        ErrorCode = "invalid escape
40     ErrInvalidNamedCapture  ErrorCode = "invalid named
41     ErrInvalidPerlOp        ErrorCode = "invalid or uns
```

```

42         ErrInvalidRepeatOp      ErrorCode = "invalid nested
43         ErrInvalidRepeatSize    ErrorCode = "invalid repeat
44         ErrInvalidUTF8          ErrorCode = "invalid UTF-8"
45         ErrMissingBracket       ErrorCode = "missing closin
46         ErrMissingParen         ErrorCode = "missing closin
47         ErrMissingRepeatArgument ErrorCode = "missing argume
48         ErrTrailingBackslash    ErrorCode = "trailing backs
49     )
50
51     func (e ErrorCode) String() string {
52         return string(e)
53     }
54
55     // Flags control the behavior of the parser and record infor
56     type Flags uint16
57
58     const (
59         FoldCase      Flags = 1 << iota // case-insensitive
60         Literal      // treat pattern as
61         ClassNL     // allow character c
62         DotNL       // allow . to match
63         OneLine     // treat ^ and $ as
64         NonGreedy   // make repetition c
65         PerlX      // allow Perl extens
66         UnicodeGroups // allow \p{Han}, \P
67         WasDollar  // regexp OpEndText
68         Simple     // regexp contains n
69
70         MatchNL = ClassNL | DotNL
71
72         Perl      = ClassNL | OneLine | PerlX | UnicodeGro
73         POSIX Flags = 0
74     )
75
76     // Pseudo-ops for parsing stack.
77     const (
78         opLeftParen = opPseudo + iota
79         opVerticalBar
80     )
81
82     type parser struct {
83         flags      Flags // parse mode flags
84         stack     []*Regexp // stack of parsed expressions
85         free      *Regexp
86         numCap    int // number of capturing groups seen
87         wholeRegexp string
88         tmpClass  []rune // temporary char class work spac
89     }
90
91     func (p *parser) newRegexp(op Op) *Regexp {

```

```

92         re := p.free
93         if re != nil {
94             p.free = re.Sub0[0]
95             *re = Regexp{ }
96         } else {
97             re = new(Regexp)
98         }
99         re.Op = op
100        return re
101    }
102
103    func (p *parser) reuse(re *Regexp) {
104        re.Sub0[0] = p.free
105        p.free = re
106    }
107
108    // Parse stack manipulation.
109
110    // push pushes the regexp re onto the parse stack and return
111    func (p *parser) push(re *Regexp) *Regexp {
112        if re.Op == OpCharClass && len(re.Rune) == 2 && re.R
113            // Single rune.
114            if p.maybeConcat(re.Rune[0], p.flags&^FoldCa
115                return nil
116        }
117        re.Op = OpLiteral
118        re.Rune = re.Rune[:1]
119        re.Flags = p.flags & FoldCase
120    } else if re.Op == OpCharClass && len(re.Rune) == 4
121        re.Rune[0] == re.Rune[1] && re.Rune[2] == re
122        unicode.SimpleFold(re.Rune[0]) == re.Rune[2]
123        unicode.SimpleFold(re.Rune[2]) == re.Rune[0]
124        re.Op == OpCharClass && len(re.Rune) == 2 &&
125            re.Rune[0]+1 == re.Rune[1] &&
126            unicode.SimpleFold(re.Rune[0]) == re
127            unicode.SimpleFold(re.Rune[1]) == re
128        // Case-insensitive rune like [Aa] or [Δδ].
129        if p.maybeConcat(re.Rune[0], p.flags|FoldCas
130            return nil
131    }
132
133        // Rewrite as (case-insensitive) literal.
134        re.Op = OpLiteral
135        re.Rune = re.Rune[:1]
136        re.Flags = p.flags | FoldCase
137    } else {
138        // Incremental concatenation.
139        p.maybeConcat(-1, 0)
140    }

```

```

141
142     p.stack = append(p.stack, re)
143     return re
144 }
145
146 // maybeConcat implements incremental concatenation
147 // of literal runes into string nodes. The parser calls thi
148 // before each push, so only the top fragment of the stack
149 // might need processing. Since this is called before a pus
150 // the topmost literal is no longer subject to operators lik
151 // (Otherwise ab* would turn into (ab)*.)
152 // If r >= 0 and there's a node left over, maybeConcat uses
153 // to push r with the given flags.
154 // maybeConcat reports whether r was pushed.
155 func (p *parser) maybeConcat(r rune, flags Flags) bool {
156     n := len(p.stack)
157     if n < 2 {
158         return false
159     }
160
161     re1 := p.stack[n-1]
162     re2 := p.stack[n-2]
163     if re1.Op != OpLiteral || re2.Op != OpLiteral || re1
164         return false
165     }
166
167     // Push re1 into re2.
168     re2.Rune = append(re2.Rune, re1.Rune...)
169
170     // Reuse re1 if possible.
171     if r >= 0 {
172         re1.Rune = re1.Rune[:1]
173         re1.Rune[0] = r
174         re1.Flags = flags
175         return true
176     }
177
178     p.stack = p.stack[:n-1]
179     p.reuse(re1)
180     return false // did not push r
181 }
182
183 // newLiteral returns a new OpLiteral Regexp with the given
184 func (p *parser) newLiteral(r rune, flags Flags) *Regexp {
185     re := p.newRegexp(OpLiteral)
186     re.Flags = flags
187     if flags&FoldCase != 0 {
188         r = minFoldRune(r)
189     }

```

```

190         re.Rune0[0] = r
191         re.Rune = re.Rune0[:1]
192         return re
193     }
194
195     // minFoldRune returns the minimum rune fold-equivalent to r
196     func minFoldRune(r rune) rune {
197         if r < minFold || r > maxFold {
198             return r
199         }
200         min := r
201         r0 := r
202         for r = unicode.SimpleFold(r); r != r0; r = unicode.
203             if min > r {
204                 min = r
205             }
206         }
207         return min
208     }
209
210     // literal pushes a literal regexp for the rune r on the sta
211     // and returns that regexp.
212     func (p *parser) literal(r rune) {
213         p.push(p.newLiteral(r, p.flags))
214     }
215
216     // op pushes a regexp with the given op onto the stack
217     // and returns that regexp.
218     func (p *parser) op(op Op) *Regexp {
219         re := p.newRegexp(op)
220         re.Flags = p.flags
221         return p.push(re)
222     }
223
224     // repeat replaces the top stack element with itself repeate
225     // before is the regexp suffix starting at the repetition op
226     // after is the regexp suffix following after the repetition
227     // repeat returns an updated 'after' and an error, if any.
228     func (p *parser) repeat(op Op, min, max int, before, after,
229         flags := p.flags
230         if p.flags&PerlX != 0 {
231             if len(after) > 0 && after[0] == '?' {
232                 after = after[1:]
233                 flags ^= NonGreedy
234             }
235             if lastRepeat != "" {
236                 // In Perl it is not allowed to stac
237                 // a** is a syntax error, not a doub
238                 // something else entirely, which we
239                 return "", &Error{ErrInvalidRepeatOp

```

```

240         }
241     }
242     n := len(p.stack)
243     if n == 0 {
244         return "", &Error{ErrMissingRepeatArgument,
245     }
246     sub := p.stack[n-1]
247     if sub.Op >= opPseudo {
248         return "", &Error{ErrMissingRepeatArgument,
249     }
250     re := p.newRegexp(op)
251     re.Min = min
252     re.Max = max
253     re.Flags = flags
254     re.Sub = re.Sub0[:1]
255     re.Sub[0] = sub
256     p.stack[n-1] = re
257     return after, nil
258 }
259
260 // concat replaces the top of the stack (above the topmost '
261 func (p *parser) concat() *Regexp {
262     p.maybeConcat(-1, 0)
263
264     // Scan down to find pseudo-operator | or (.
265     i := len(p.stack)
266     for i > 0 && p.stack[i-1].Op < opPseudo {
267         i--
268     }
269     subs := p.stack[i:]
270     p.stack = p.stack[:i]
271
272     // Empty concatenation is special case.
273     if len(subs) == 0 {
274         return p.push(p.newRegexp(OpEmptyMatch))
275     }
276
277     return p.push(p.collapse(subs, OpConcat))
278 }
279
280 // alternate replaces the top of the stack (above the topmos
281 func (p *parser) alternate() *Regexp {
282     // Scan down to find pseudo-operator (.
283     // There are no | above (.
284     i := len(p.stack)
285     for i > 0 && p.stack[i-1].Op < opPseudo {
286         i--
287     }
288     subs := p.stack[i:]

```

```

289         p.stack = p.stack[:i]
290
291         // Make sure top class is clean.
292         // All the others already are (see swapVerticalBar).
293         if len(subs) > 0 {
294             cleanAlt(subs[len(subs)-1])
295         }
296
297         // Empty alternate is special case
298         // (shouldn't happen but easy to handle).
299         if len(subs) == 0 {
300             return p.push(p.newRegexp(OpNoMatch))
301         }
302
303         return p.push(p.collapse(subs, OpAlternate))
304     }
305
306     // cleanAlt cleans re for eventual inclusion in an alternati
307     func cleanAlt(re *Regexp) {
308         switch re.Op {
309         case OpCharClass:
310             re.Rune = cleanClass(&re.Rune)
311             if len(re.Rune) == 2 && re.Rune[0] == 0 && r
312                 re.Rune = nil
313                 re.Op = OpAnyChar
314                 return
315             }
316             if len(re.Rune) == 4 && re.Rune[0] == 0 && r
317                 re.Rune = nil
318                 re.Op = OpAnyCharNotNL
319                 return
320             }
321             if cap(re.Rune)-len(re.Rune) > 100 {
322                 // re.Rune will not grow any more.
323                 // Make a copy or inline to reclaim
324                 re.Rune = append(re.Rune[:0], re.Ru
325             }
326         }
327     }
328
329     // collapse returns the result of applying op to sub.
330     // If sub contains op nodes, they all get hoisted up
331     // so that there is never a concat of a concat or an
332     // alternate of an alternate.
333     func (p *parser) collapse(subs []*Regexp, op Op) *Regexp {
334         if len(subs) == 1 {
335             return subs[0]
336         }
337         re := p.newRegexp(op)

```

```

338     re.Sub = re.Sub0[:0]
339     for _, sub := range subs {
340         if sub.Op == op {
341             re.Sub = append(re.Sub, sub.Sub...)
342             p.reuse(sub)
343         } else {
344             re.Sub = append(re.Sub, sub)
345         }
346     }
347     if op == OpAlternate {
348         re.Sub = p.factor(re.Sub, re.Flags)
349         if len(re.Sub) == 1 {
350             old := re
351             re = re.Sub[0]
352             p.reuse(old)
353         }
354     }
355     return re
356 }
357
358 // factor factors common prefixes from the alternation list
359 // It returns a replacement list that reuses the same storage
360 // frees (passes to p.reuse) any removed *Regexp.
361 //
362 // For example,
363 //     ABC|ABD|AEF|BCX|BCY
364 // simplifies by literal prefix extraction to
365 //     A(B(C|D)|EF)|BC(X|Y)
366 // which simplifies by character class introduction to
367 //     A(B[CD]|EF)|BC[XY]
368 //
369 func (p *parser) factor(sub []*Regexp, flags Flags) []*Regexp
370     if len(sub) < 2 {
371         return sub
372     }
373
374     // Round 1: Factor out common literal prefixes.
375     var str []rune
376     var strflags Flags
377     start := 0
378     out := sub[:0]
379     for i := 0; i <= len(sub); i++ {
380         // Invariant: the Regexp that were in sub[0]
381         // used or marked for reuse, and the slice s
382         // for out (len(out) <= start).
383         //
384         // Invariant: sub[start:i] consists of regexp
385         // with str as modified by strflags.
386         var istr []rune
387         var iflags Flags

```

```

388         if i < len(sub) {
389             istr, iflags = p.leadingString(sub[i]
390             if iflags == strflags {
391                 same := 0
392                 for same < len(str) && same
393                     same++
394             }
395             if same > 0 {
396                 // Matches at least
397                 // Keep going around
398                 str = str[:same]
399                 continue
400             }
401         }
402     }
403
404     // Found end of a run with common leading li
405     // sub[start:i] all begin with str[0:len(str)
406     // does not even begin with str[0].
407     //
408     // Factor out common string and append facto
409     if i == start {
410         // Nothing to do - run of length 0.
411     } else if i == start+1 {
412         // Just one: don't bother factoring.
413         out = append(out, sub[start])
414     } else {
415         // Construct factored form: prefix(s
416         prefix := p.newRegexp(OpLiteral)
417         prefix.Flags = strflags
418         prefix.Rune = append(prefix.Rune[:0]
419
420         for j := start; j < i; j++ {
421             sub[j] = p.removeLeadingStri
422         }
423         suffix := p.collapse(sub[start:i], 0
424
425         re := p.newRegexp(OpConcat)
426         re.Sub = append(re.Sub[:0], prefix,
427         out = append(out, re)
428     }
429
430     // Prepare for next iteration.
431     start = i
432     str = istr
433     strflags = iflags
434 }
435 sub = out
436

```

```

437 // Round 2: Factor out common complex prefixes,
438 // just the first piece of each concatenation,
439 // whatever it is. This is good enough a lot of the
440 start = 0
441 out = sub[:0]
442 var first *Regexp
443 for i := 0; i <= len(sub); i++ {
444     // Invariant: the Regexps that were in sub[0
445     // used or marked for reuse, and the slice s
446     // for out (len(out) <= start).
447     //
448     // Invariant: sub[start:i] consists of regex
449     var ifirst *Regexp
450     if i < len(sub) {
451         ifirst = p.leadingRegexp(sub[i])
452         if first != nil && first.Equal(ifirs
453             continue
454     }
455 }
456
457 // Found end of a run with common leading re
458 // sub[start:i] all begin with first but sub
459 //
460 // Factor out common regexp and append facto
461 if i == start {
462     // Nothing to do - run of length 0.
463 } else if i == start+1 {
464     // Just one: don't bother factoring.
465     out = append(out, sub[start])
466 } else {
467     // Construct factored form: prefix(s
468     prefix := first
469     for j := start; j < i; j++ {
470         reuse := j != start // prefi
471         sub[j] = p.removeLeadingRege
472     }
473     suffix := p.collapse(sub[start:i], 0
474
475     re := p.newRegexp(OpConcat)
476     re.Sub = append(re.Sub[:0], prefix,
477     out = append(out, re)
478 }
479
480 // Prepare for next iteration.
481 start = i
482 first = ifirst
483 }
484 sub = out
485

```

```

486 // Round 3: Collapse runs of single literals into ch
487 start = 0
488 out = sub[:0]
489 for i := 0; i <= len(sub); i++ {
490     // Invariant: the Regexps that were in sub[0
491     // used or marked for reuse, and the slice s
492     // for out (len(out) <= start).
493     //
494     // Invariant: sub[start:i] consists of regex
495     // literal runes or character classes.
496     if i < len(sub) && isCharClass(sub[i]) {
497         continue
498     }
499
500     // sub[i] is not a char or char class;
501     // emit char class for sub[start:i]...
502     if i == start {
503         // Nothing to do - run of length 0.
504     } else if i == start+1 {
505         out = append(out, sub[start])
506     } else {
507         // Make new char class.
508         // Start with most complex regexp in
509         max := start
510         for j := start + 1; j < i; j++ {
511             if sub[max].Op < sub[j].Op |
512                 max = j
513         }
514     }
515     sub[start], sub[max] = sub[max], sub
516
517     for j := start + 1; j < i; j++ {
518         mergeCharClass(sub[start], s
519             p.reuse(sub[j])
520     }
521     cleanAlt(sub[start])
522     out = append(out, sub[start])
523 }
524
525 // ... and then emit sub[i].
526 if i < len(sub) {
527     out = append(out, sub[i])
528 }
529 start = i + 1
530 }
531 sub = out
532
533 // Round 4: Collapse runs of empty matches into a si
534 start = 0
535 out = sub[:0]

```

```

536         for i := range sub {
537             if i+1 < len(sub) && sub[i].Op == OpEmptyMat
538                 continue
539         }
540         out = append(out, sub[i])
541     }
542     sub = out
543
544     return sub
545 }
546
547 // leadingString returns the leading literal string that re
548 // The string refers to storage in re or its children.
549 func (p *parser) leadingString(re *Regexp) ([]rune, Flags) {
550     if re.Op == OpConcat && len(re.Sub) > 0 {
551         re = re.Sub[0]
552     }
553     if re.Op != OpLiteral {
554         return nil, 0
555     }
556     return re.Rune, re.Flags & FoldCase
557 }
558
559 // removeLeadingString removes the first n leading runes
560 // from the beginning of re. It returns the replacement for
561 func (p *parser) removeLeadingString(re *Regexp, n int) *Reg
562     if re.Op == OpConcat && len(re.Sub) > 0 {
563         // Removing a leading string in a concatenat
564         // might simplify the concatenation.
565         sub := re.Sub[0]
566         sub = p.removeLeadingString(sub, n)
567         re.Sub[0] = sub
568         if sub.Op == OpEmptyMatch {
569             p.reuse(sub)
570             switch len(re.Sub) {
571             case 0, 1:
572                 // Impossible but handle.
573                 re.Op = OpEmptyMatch
574                 re.Sub = nil
575             case 2:
576                 old := re
577                 re = re.Sub[1]
578                 p.reuse(old)
579             default:
580                 copy(re.Sub, re.Sub[1:])
581                 re.Sub = re.Sub[:len(re.Sub)]
582             }
583         }
584     }
    return re

```

```

585     }
586
587     if re.Op == OpLiteral {
588         re.Rune = re.Rune[:copy(re.Rune, re.Rune[n:])]
589         if len(re.Rune) == 0 {
590             re.Op = OpEmptyMatch
591         }
592     }
593     return re
594 }
595
596 // leadingRegex returns the leading regexp that re begins w
597 // The regexp refers to storage in re or its children.
598 func (p *parser) leadingRegex(re *Regex) *Regex {
599     if re.Op == OpEmptyMatch {
600         return nil
601     }
602     if re.Op == OpConcat && len(re.Sub) > 0 {
603         sub := re.Sub[0]
604         if sub.Op == OpEmptyMatch {
605             return nil
606         }
607         return sub
608     }
609     return re
610 }
611
612 // removeLeadingRegex removes the leading regexp in re.
613 // It returns the replacement for re.
614 // If reuse is true, it passes the removed regexp (if no lon
615 func (p *parser) removeLeadingRegex(re *Regex, reuse bool)
616     if re.Op == OpConcat && len(re.Sub) > 0 {
617         if reuse {
618             p.reuse(re.Sub[0])
619         }
620         re.Sub = re.Sub[:copy(re.Sub, re.Sub[1:])]
621         switch len(re.Sub) {
622         case 0:
623             re.Op = OpEmptyMatch
624             re.Sub = nil
625         case 1:
626             old := re
627             re = re.Sub[0]
628             p.reuse(old)
629         }
630         return re
631     }
632     if reuse {
633         p.reuse(re)

```

```

634     }
635     return p.newRegexp(OpEmptyMatch)
636 }
637
638 func literalRegexp(s string, flags Flags) *Regexp {
639     re := &Regexp{Op: OpLiteral}
640     re.Flags = flags
641     re.Rune = re.Rune0[:0] // use local storage for small
642     for _, c := range s {
643         if len(re.Rune) >= cap(re.Rune) {
644             // string is too long to fit in Rune
645             re.Rune = []rune(s)
646             break
647         }
648         re.Rune = append(re.Rune, c)
649     }
650     return re
651 }
652
653 // Parsing.
654
655 // Parse parses a regular expression string s, controlled by
656 // Flags, and returns a regular expression parse tree. The s
657 // described in the top-level comment for package regexp.
658 func Parse(s string, flags Flags) (*Regexp, error) {
659     if flags&Literal != 0 {
660         // Trivial parser for literal string.
661         if err := checkUTF8(s); err != nil {
662             return nil, err
663         }
664         return literalRegexp(s, flags), nil
665     }
666
667     // Otherwise, must do real work.
668     var (
669         p          parser
670         err         error
671         c           rune
672         op          Op
673         lastRepeat string
674         min, max    int
675     )
676     p.flags = flags
677     p.wholeRegexp = s
678     t := s
679     for t != "" {
680         repeat := ""
681     BigSwitch:
682         switch t[0] {
683         default:

```

```

684         if c, t, err = nextRune(t); err != n
685             return nil, err
686     }
687     p.literal(c)
688
689     case '(':
690         if p.flags&PerlX != 0 && len(t) >= 2
691             // Flag changes and non-capt
692             if t, err = p.parsePerlFlags
693                 return nil, err
694             }
695         break
696     }
697     p.numCap++
698     p.op(opLeftParen).Cap = p.numCap
699     t = t[1:]
700     case '|':
701         if err = p.parseVerticalBar(); err !=
702             return nil, err
703     }
704     t = t[1:]
705     case ')':
706         if err = p.parseRightParen(); err !=
707             return nil, err
708     }
709     t = t[1:]
710     case '^':
711         if p.flags&OneLine != 0 {
712             p.op(OpBeginText)
713         } else {
714             p.op(OpBeginLine)
715         }
716     t = t[1:]
717     case '$':
718         if p.flags&OneLine != 0 {
719             p.op(OpEndText).Flags |= Was
720         } else {
721             p.op(OpEndLine)
722         }
723     t = t[1:]
724     case '.':
725         if p.flags&DotNL != 0 {
726             p.op(OpAnyChar)
727         } else {
728             p.op(OpAnyCharNotNL)
729         }
730     t = t[1:]
731     case '[':
732         if t, err = p.parseClass(t); err !=

```

```

733         return nil, err
734     }
735     case '*', '+', '?':
736         before := t
737         switch t[0] {
738             case '*':
739                 op = OpStar
740             case '+':
741                 op = OpPlus
742             case '?':
743                 op = OpQuest
744         }
745         after := t[1:]
746         if after, err = p.repeat(op, min, ma
747             return nil, err
748         }
749         repeat = before
750         t = after
751     case '{':
752         op = OpRepeat
753         before := t
754         min, max, after, ok := p.parseRepeat
755         if !ok {
756             // If the repeat cannot be p
757             p.literal('{')
758             t = t[1:]
759             break
760         }
761         if min < 0 || min > 1000 || max > 10
762             // Numbers were too big, or
763             return nil, &Error{ErrInvali
764         }
765         if after, err = p.repeat(op, min, ma
766             return nil, err
767         }
768         repeat = before
769         t = after
770     case '\\':
771         if p.flags&PerlX != 0 && len(t) >= 2
772             switch t[1] {
773                 case 'A':
774                     p.op(OpBeginText)
775                     t = t[2:]
776                     break BigSwitch
777                 case 'b':
778                     p.op(OpWordBoundary)
779                     t = t[2:]
780                     break BigSwitch
781                 case 'B':

```

```

782             p.op(OpNoWordBoundar
783             t = t[2:]
784             break BigSwitch
785         case 'C':
786             // any byte; not sup
787             return nil, &Error{E
788         case 'Q':
789             // \Q ... \E: the ..
790             var lit string
791             if i := strings.Inde
792                 lit = t[2:]
793                 t = ""
794             } else {
795                 lit = t[2:i]
796                 t = t[i+2:]
797             }
798             p.push(literalRegex
799             break BigSwitch
800         case 'z':
801             p.op(OpEndText)
802             t = t[2:]
803             break BigSwitch
804     }
805 }
806
807 re := p.newRegexp(OpCharClass)
808 re.Flags = p.flags
809
810 // Look for Unicode character group
811 if len(t) >= 2 && (t[1] == 'p' || t[
812     r, rest, err := p.parseUnico
813     if err != nil {
814         return nil, err
815     }
816     if r != nil {
817         re.Rune = r
818         t = rest
819         p.push(re)
820         break BigSwitch
821     }
822 }
823
824 // Perl character class escape.
825 if r, rest := p.parsePerlClassEscape
826     re.Rune = r
827     t = rest
828     p.push(re)
829     break BigSwitch
830 }
831 p.reuse(re)

```

```

832
833             // Ordinary single-character escape.
834             if c, t, err = p.parseEscape(t); err
835                 return nil, err
836             }
837             p.literal(c)
838         }
839         lastRepeat = repeat
840     }
841
842     p.concat()
843     if p.swapVerticalBar() {
844         // pop vertical bar
845         p.stack = p.stack[:len(p.stack)-1]
846     }
847     p.alternate()
848
849     n := len(p.stack)
850     if n != 1 {
851         return nil, &Error{ErrMissingParen, s}
852     }
853     return p.stack[0], nil
854 }
855
856 // parseRepeat parses {min} (max=min) or {min,} (max=-1) or
857 // If s is not of that form, it returns ok == false.
858 // If s has the right form but the values are too big, it re
859 func (p *parser) parseRepeat(s string) (min, max int, rest s
860     if s == "" || s[0] != '{' {
861         return
862     }
863     s = s[1:]
864     var ok1 bool
865     if min, s, ok1 = p.parseInt(s); !ok1 {
866         return
867     }
868     if s == "" {
869         return
870     }
871     if s[0] != ',' {
872         max = min
873     } else {
874         s = s[1:]
875         if s == "" {
876             return
877         }
878         if s[0] == '}' {
879             max = -1
880         } else if max, s, ok1 = p.parseInt(s); !ok1

```

```

881         return
882     } else if max < 0 {
883         // parseInt found too big a number
884         min = -1
885     }
886 }
887 if s == "" || s[0] != '}' {
888     return
889 }
890 rest = s[1:]
891 ok = true
892 return
893 }
894
895 // parsePerlFlags parses a Perl flag setting or non-capturing
896 // like (?i) or (?: or (?i:. It removes the prefix from s and
897 // The caller must have ensured that s begins with "(?".
898 func (p *parser) parsePerlFlags(s string) (rest string, err
899     t := s
900
901     // Check for named captures, first introduced in Python
902     // As usual, there are three slightly different synt
903     //
904     // (?P<name>expr) the original, introduced by Python
905     // (?<name>expr) the .NET alteration, adopted by
906     // (?'name'expr) another .NET alteration, adopted
907     //
908     // Perl 5.10 gave in and implemented the Python version
909     // but they claim that the last two are the preferred
910     // PCRE and languages based on it (specifically, PHP
911     // support all three as well. EcmaScript 4 uses only
912     //
913     // In both the open source world (via Code Search) and
914     // Google source tree, (?P<expr>name) is the dominant
915     // so that's the one we implement. One is enough.
916     if len(t) > 4 && t[2] == 'P' && t[3] == '<' {
917         // Pull out name.
918         end := strings.IndexRune(t, '>')
919         if end < 0 {
920             if err = checkUTF8(t); err != nil {
921                 return "", err
922             }
923             return "", &Error{ErrInvalidNamedCapture}
924         }
925
926         capture := t[:end+1] // "(?P<name>"
927         name := t[4:end]     // "name"
928         if err = checkUTF8(name); err != nil {
929             return "", err

```

```

930     }
931     if !isValidCaptureName(name) {
932         return "", &Error{ErrInvalidNamedCap
933     }
934
935     // Like ordinary capture, but named.
936     p.numCap++
937     re := p.op(opLeftParen)
938     re.Cap = p.numCap
939     re.Name = name
940     return t[end+1:], nil
941 }
942
943 // Non-capturing group. Might also twiddle Perl fla
944 var c rune
945 t = t[2:] // skip (?
946 flags := p.flags
947 sign := +1
948 sawFlag := false
949 Loop:
950     for t != "" {
951         if c, t, err = nextRune(t); err != nil {
952             return "", err
953         }
954         switch c {
955         default:
956             break Loop
957
958         // Flags.
959         case 'i':
960             flags |= FoldCase
961             sawFlag = true
962         case 'm':
963             flags &^= OneLine
964             sawFlag = true
965         case 's':
966             flags |= DotNL
967             sawFlag = true
968         case 'U':
969             flags |= NonGreedy
970             sawFlag = true
971
972         // Switch to negation.
973         case '-':
974             if sign < 0 {
975                 break Loop
976             }
977             sign = -1
978             // Invert flags so that | above turn
979             // We'll invert flags again before u

```

```

980             flags = ^flags
981             sawFlag = false
982
983             // End of flags, starting group or not.
984             case ':', ')':
985                 if sign < 0 {
986                     if !sawFlag {
987                         break Loop
988                     }
989                     flags = ^flags
990                 }
991                 if c == ':' {
992                     // Open new group
993                     p.op(opLeftParen)
994                 }
995                 p.flags = flags
996                 return t, nil
997             }
998         }
999
1000         return "", &Error{ErrInvalidPerlOp, s[:len(s)-len(t)}
1001     }
1002
1003     // isValidCaptureName reports whether name
1004     // is a valid capture name: [A-Za-z0-9_]+.
1005     // PCRE limits names to 32 bytes.
1006     // Python rejects names starting with digits.
1007     // We don't enforce either of those.
1008     func isValidCaptureName(name string) bool {
1009         if name == "" {
1010             return false
1011         }
1012         for _, c := range name {
1013             if c != '_' && !isalnum(c) {
1014                 return false
1015             }
1016         }
1017         return true
1018     }
1019
1020     // parseInt parses a decimal integer.
1021     func (p *parser) parseInt(s string) (n int, rest string, ok
1022         if s == "" || s[0] < '0' || '9' < s[0] {
1023             return
1024         }
1025         // Disallow leading zeros.
1026         if len(s) >= 2 && s[0] == '0' && '0' <= s[1] && s[1]
1027             return
1028     }

```

```

1029         t := s
1030         for s != "" && '0' <= s[0] && s[0] <= '9' {
1031             s = s[1:]
1032         }
1033         rest = s
1034         ok = true
1035         // Have digits, compute value.
1036         t = t[:len(t)-len(s)]
1037         for i := 0; i < len(t); i++ {
1038             // Avoid overflow.
1039             if n >= 1e8 {
1040                 n = -1
1041                 break
1042             }
1043             n = n*10 + int(t[i]) - '0'
1044         }
1045         return
1046     }
1047
1048     // can this be represented as a character class?
1049     // single-rune literal string, char class, ., and .|\n.
1050     func isCharClass(re *Regexp) bool {
1051         return re.Op == OpLiteral && len(re.Rune) == 1 ||
1052             re.Op == OpCharClass ||
1053             re.Op == OpAnyCharNotNL ||
1054             re.Op == OpAnyChar
1055     }
1056
1057     // does re match r?
1058     func matchRune(re *Regexp, r rune) bool {
1059         switch re.Op {
1060         case OpLiteral:
1061             return len(re.Rune) == 1 && re.Rune[0] == r
1062         case OpCharClass:
1063             for i := 0; i < len(re.Rune); i += 2 {
1064                 if re.Rune[i] <= r && r <= re.Rune[i+1]
1065                     return true
1066             }
1067         }
1068         return false
1069     case OpAnyCharNotNL:
1070         return r != '\n'
1071     case OpAnyChar:
1072         return true
1073     }
1074     return false
1075 }
1076
1077 // parseVerticalBar handles a | in the input.

```

```

1078 func (p *parser) parseVerticalBar() error {
1079     p.concat()
1080
1081     // The concatenation we just parsed is on top of the
1082     // If it sits above an opVerticalBar, swap it below
1083     // (things below an opVerticalBar become an alternat
1084     // Otherwise, push a new vertical bar.
1085     if !p.swapVerticalBar() {
1086         p.op(opVerticalBar)
1087     }
1088
1089     return nil
1090 }
1091
1092 // mergeCharClass makes dst = dst|src.
1093 // The caller must ensure that dst.Op >= src.Op,
1094 // to reduce the amount of copying.
1095 func mergeCharClass(dst, src *Regexp) {
1096     switch dst.Op {
1097     case OpAnyChar:
1098         // src doesn't add anything.
1099     case OpAnyCharNotNL:
1100         // src might add \n
1101         if matchRune(src, '\n') {
1102             dst.Op = OpAnyChar
1103         }
1104     case OpCharClass:
1105         // src is simpler, so either literal or char
1106         if src.Op == OpLiteral {
1107             dst.Rune = appendLiteral(dst.Rune, s
1108         } else {
1109             dst.Rune = appendClass(dst.Rune, src
1110         }
1111     case OpLiteral:
1112         // both literal
1113         if src.Rune[0] == dst.Rune[0] && src.Flags =
1114             break
1115         }
1116         dst.Op = OpCharClass
1117         dst.Rune = appendLiteral(dst.Rune[:0], dst.R
1118         dst.Rune = appendLiteral(dst.Rune, src.Rune[
1119     }
1120 }
1121
1122 // If the top of the stack is an element followed by an opVe
1123 // swapVerticalBar swaps the two and returns true.
1124 // Otherwise it returns false.
1125 func (p *parser) swapVerticalBar() bool {
1126     // If above and below vertical bar are literal or ch
1127     // can merge into a single char class.

```

```

1128     n := len(p.stack)
1129     if n >= 3 && p.stack[n-2].Op == opVerticalBar && isC
1130         re1 := p.stack[n-1]
1131         re3 := p.stack[n-3]
1132         // Make re3 the more complex of the two.
1133         if re1.Op > re3.Op {
1134             re1, re3 = re3, re1
1135             p.stack[n-3] = re3
1136         }
1137         mergeCharClass(re3, re1)
1138         p.reuse(re1)
1139         p.stack = p.stack[:n-1]
1140         return true
1141     }
1142
1143     if n >= 2 {
1144         re1 := p.stack[n-1]
1145         re2 := p.stack[n-2]
1146         if re2.Op == opVerticalBar {
1147             if n >= 3 {
1148                 // Now out of reach.
1149                 // Clean opportunistically.
1150                 cleanAlt(p.stack[n-3])
1151             }
1152             p.stack[n-2] = re1
1153             p.stack[n-1] = re2
1154             return true
1155         }
1156     }
1157     return false
1158 }
1159
1160 // parseRightParen handles a ) in the input.
1161 func (p *parser) parseRightParen() error {
1162     p.concat()
1163     if p.swapVerticalBar() {
1164         // pop vertical bar
1165         p.stack = p.stack[:len(p.stack)-1]
1166     }
1167     p.alternate()
1168
1169     n := len(p.stack)
1170     if n < 2 {
1171         return &Error{ErrInternalError, ""}
1172     }
1173     re1 := p.stack[n-1]
1174     re2 := p.stack[n-2]
1175     p.stack = p.stack[:n-2]
1176     if re2.Op != opLeftParen {

```

```

1177         return &Error{ErrMissingParen, p.wholeRegexp
1178     }
1179     // Restore flags at time of paren.
1180     p.flags = re2.Flags
1181     if re2.Cap == 0 {
1182         // Just for grouping.
1183         p.push(re1)
1184     } else {
1185         re2.Op = OpCapture
1186         re2.Sub = re2.Sub0[:1]
1187         re2.Sub[0] = re1
1188         p.push(re2)
1189     }
1190     return nil
1191 }
1192
1193 // parseEscape parses an escape sequence at the beginning of
1194 // and returns the rune.
1195 func (p *parser) parseEscape(s string) (r rune, rest string,
1196     t := s[1:]
1197     if t == "" {
1198         return 0, "", &Error{ErrTrailingBackslash, "
1199     }
1200     c, t, err := nextRune(t)
1201     if err != nil {
1202         return 0, "", err
1203     }
1204
1205 Switch:
1206     switch c {
1207     default:
1208         if c < utf8.RuneSelf && !isalnum(c) {
1209             // Escaped non-word characters are a
1210             // PCRE is not quite so rigorous: it
1211             // \q, but we don't. We once reject
1212             // programs and people insist on usi
1213             return c, t, nil
1214         }
1215
1216         // Octal escapes.
1217         case '1', '2', '3', '4', '5', '6', '7':
1218             // Single non-zero digit is a backreference;
1219             if t == "" || t[0] < '0' || t[0] > '7' {
1220                 break
1221             }
1222             fallthrough
1223         case '0':
1224             // Consume up to three octal digits; already
1225             r = c - '0'

```

```

1226         for i := 1; i < 3; i++ {
1227             if t == "" || t[0] < '0' || t[0] > '
1228                 break
1229             }
1230             r = r*8 + rune(t[0]) - '0'
1231             t = t[1:]
1232         }
1233         return r, t, nil
1234
1235     // Hexadecimal escapes.
1236     case 'x':
1237         if t == "" {
1238             break
1239         }
1240         if c, t, err = nextRune(t); err != nil {
1241             return 0, "", err
1242         }
1243         if c == '{' {
1244             // Any number of digits in braces.
1245             // Perl accepts any text at all; it
1246             // after the first non-hex digit. w
1247             // and at least one.
1248             nhex := 0
1249             r = 0
1250             for {
1251                 if t == "" {
1252                     break Switch
1253                 }
1254                 if c, t, err = nextRune(t);
1255                     return 0, "", err
1256                 }
1257                 if c == '}' {
1258                     break
1259                 }
1260                 v := unhex(c)
1261                 if v < 0 {
1262                     break Switch
1263                 }
1264                 r = r*16 + v
1265                 if r > unicode.MaxRune {
1266                     break Switch
1267                 }
1268                 nhex++
1269             }
1270             if nhex == 0 {
1271                 break Switch
1272             }
1273             return r, t, nil
1274         }
1275

```

```

1276         // Easy case: two hex digits.
1277         x := unhex(c)
1278         if c, t, err = nextRune(t); err != nil {
1279             return 0, "", err
1280         }
1281         y := unhex(c)
1282         if x < 0 || y < 0 {
1283             break
1284         }
1285         return x*16 + y, t, nil
1286
1287         // C escapes. There is no case 'b', to avoid mispar
1288         // the Perl word-boundary \b as the C backspace \b
1289         // when in POSIX mode. In Perl, /\b/ means word-bou
1290         // but /[\\b]/ means backspace. We don't support tha
1291         // If you want a backspace, embed a literal backspac
1292         // character or use \x08.
1293         case 'a':
1294             return '\a', t, err
1295         case 'f':
1296             return '\f', t, err
1297         case 'n':
1298             return '\n', t, err
1299         case 'r':
1300             return '\r', t, err
1301         case 't':
1302             return '\t', t, err
1303         case 'v':
1304             return '\v', t, err
1305         }
1306         return 0, "", &Error{ErrInvalidEscape, s[:len(s)-len
1307     }
1308
1309     // parseClassChar parses a character class character at the
1310     // and returns it.
1311     func (p *parser) parseClassChar(s, wholeClass string) (r run
1312         if s == "" {
1313             return 0, "", &Error{Code: ErrMissingBracket
1314         }
1315
1316         // Allow regular escape sequences even though
1317         // many need not be escaped in this context.
1318         if s[0] == '\\' {
1319             return p.parseEscape(s)
1320         }
1321
1322         return nextRune(s)
1323     }
1324

```

```

1325 type charGroup struct {
1326     sign int
1327     class []rune
1328 }
1329
1330 // parsePerlClassEscape parses a leading Perl character clas
1331 // from the beginning of s. If one is present, it appends t
1332 // and returns the new slice r and the remainder of the stri
1333 func (p *parser) parsePerlClassEscape(s string, r []rune) (o
1334     if p.flags&PerlX == 0 || len(s) < 2 || s[0] != '\\'
1335         return
1336     }
1337     g := perlGroup[s[0:2]]
1338     if g.sign == 0 {
1339         return
1340     }
1341     return p.appendGroup(r, g), s[2:]
1342 }
1343
1344 // parseNamedClass parses a leading POSIX named character cl
1345 // from the beginning of s. If one is present, it appends t
1346 // and returns the new slice r and the remainder of the stri
1347 func (p *parser) parseNamedClass(s string, r []rune) (out []
1348     if len(s) < 2 || s[0] != '[' || s[1] != ':' {
1349         return
1350     }
1351
1352     i := strings.Index(s[2:], ":]")
1353     if i < 0 {
1354         return
1355     }
1356     i += 2
1357     name, s := s[0:i+2], s[i+2:]
1358     g := posixGroup[name]
1359     if g.sign == 0 {
1360         return nil, "", &Error{ErrInvalidCharRange,
1361     }
1362     return p.appendGroup(r, g), s, nil
1363 }
1364
1365 func (p *parser) appendGroup(r []rune, g charGroup) []rune {
1366     if p.flags&FoldCase == 0 {
1367         if g.sign < 0 {
1368             r = appendNegatedClass(r, g.class)
1369         } else {
1370             r = appendClass(r, g.class)
1371         }
1372     } else {
1373         tmp := p.tmpClass[:0]

```

```

1374         tmp = appendFoldedClass(tmp, g.class)
1375         p.tmpClass = tmp
1376         tmp = cleanClass(&p.tmpClass)
1377         if g.sign < 0 {
1378             r = appendNegatedClass(r, tmp)
1379         } else {
1380             r = appendClass(r, tmp)
1381         }
1382     }
1383     return r
1384 }
1385
1386 var anyTable = &unicode.RangeTable{
1387     R16: []unicode.Range16{{Lo: 0, Hi: 1<<16 - 1, Stride
1388     R32: []unicode.Range32{{Lo: 1 << 16, Hi: unicode.Max
1389 }
1390
1391 // unicodeTable returns the unicode.RangeTable identified by
1392 // and the table of additional fold-equivalent code points.
1393 func unicodeTable(name string) (*unicode.RangeTable, *unicod
1394     // Special case: "Any" means any.
1395     if name == "Any" {
1396         return anyTable, anyTable
1397     }
1398     if t := unicode.Categories[name]; t != nil {
1399         return t, unicode.FoldCategory[name]
1400     }
1401     if t := unicode.Scripts[name]; t != nil {
1402         return t, unicode.FoldScript[name]
1403     }
1404     return nil, nil
1405 }
1406
1407 // parseUnicodeClass parses a leading Unicode character clas
1408 // from the beginning of s. If one is present, it appends t
1409 // and returns the new slice r and the remainder of the stri
1410 func (p *parser) parseUnicodeClass(s string, r []rune) (out
1411     if p.flags&UnicodeGroups == 0 || len(s) < 2 || s[0]
1412         return
1413     }
1414
1415     // Committed to parse or return error.
1416     sign := +1
1417     if s[1] == 'P' {
1418         sign = -1
1419     }
1420     t := s[2:]
1421     c, t, err := nextRune(t)
1422     if err != nil {
1423         return

```

```

1424     }
1425     var seq, name string
1426     if c != '{' {
1427         // Single-letter name.
1428         seq = s[:len(s)-len(t)]
1429         name = seq[2:]
1430     } else {
1431         // Name is in braces.
1432         end := strings.IndexRune(s, '}')
1433         if end < 0 {
1434             if err = checkUTF8(s); err != nil {
1435                 return
1436             }
1437             return nil, "", &Error{ErrInvalidCha
1438         }
1439         seq, t = s[:end+1], s[end+1:]
1440         name = s[3:end]
1441         if err = checkUTF8(name); err != nil {
1442             return
1443         }
1444     }
1445
1446     // Group can have leading negation too. \p{^Han} ==
1447     if name != "" && name[0] == '^' {
1448         sign = -sign
1449         name = name[1:]
1450     }
1451
1452     tab, fold := unicodeTable(name)
1453     if tab == nil {
1454         return nil, "", &Error{ErrInvalidCharRange,
1455     }
1456
1457     if p.flags&FoldCase == 0 || fold == nil {
1458         if sign > 0 {
1459             r = appendTable(r, tab)
1460         } else {
1461             r = appendNegatedTable(r, tab)
1462         }
1463     } else {
1464         // Merge and clean tab and fold in a tempora
1465         // This is necessary for the negative case a
1466         // for the positive case.
1467         tmp := p.tmpClass[:0]
1468         tmp = appendTable(tmp, tab)
1469         tmp = appendTable(tmp, fold)
1470         p.tmpClass = tmp
1471         tmp = cleanClass(&p.tmpClass)
1472         if sign > 0 {

```

```

1473         r = appendClass(r, tmp)
1474     } else {
1475         r = appendNegatedClass(r, tmp)
1476     }
1477 }
1478 return r, t, nil
1479 }
1480
1481 // parseClass parses a character class at the beginning of s
1482 // and pushes it onto the parse stack.
1483 func (p *parser) parseClass(s string) (rest string, err error) {
1484     t := s[1:] // chop [
1485     re := p.newRegexp(OpCharClass)
1486     re.Flags = p.flags
1487     re.Rune = re.Rune0[:0]
1488
1489     sign := +1
1490     if t != "" && t[0] == '^' {
1491         sign = -1
1492         t = t[1:]
1493
1494         // If character class does not match \n, add
1495         // so that negation later will do the right
1496         if p.flags&ClassNL == 0 {
1497             re.Rune = append(re.Rune, '\n', '\n'
1498         )
1499     }
1500
1501     class := re.Rune
1502     first := true // ] and - are okay as first char in c
1503     for t != "" || t[0] != ']' || first {
1504         // POSIX: - is only okay unescaped as first
1505         // Perl: - is okay anywhere.
1506         if t != "" && t[0] == '-' && p.flags&PerlX =
1507             _, size := utf8.DecodeRuneInString(t
1508             return "", &Error{Code: ErrInvalidCh
1509         }
1510         first = false
1511
1512         // Look for POSIX [:alnum:] etc.
1513         if len(t) > 2 && t[0] == '[' && t[1] == ':'
1514             nclass, nt, err := p.parseNamedClass
1515             if err != nil {
1516                 return "", err
1517             }
1518             if nclass != nil {
1519                 class, t = nclass, nt
1520                 continue
1521         }

```

```

1522     }
1523
1524     // Look for Unicode character group like \p{
1525     nclass, nt, err := p.parseUnicodeClass(t, cl
1526     if err != nil {
1527         return "", err
1528     }
1529     if nclass != nil {
1530         class, t = nclass, nt
1531         continue
1532     }
1533
1534     // Look for Perl character class symbols (ex
1535     if nclass, nt := p.parsePerlClassEscape(t, c
1536         class, t = nclass, nt
1537         continue
1538     }
1539
1540     // Single character or simple range.
1541     rng := t
1542     var lo, hi rune
1543     if lo, t, err = p.parseClassChar(t, s); err
1544         return "", err
1545     }
1546     hi = lo
1547     // [a-] means (a|-) so check for final ].
1548     if len(t) >= 2 && t[0] == '-' && t[1] != ']'
1549         t = t[1:]
1550         if hi, t, err = p.parseClassChar(t,
1551             return "", err
1552         }
1553         if hi < lo {
1554             rng = rng[:len(rng)-len(t)]
1555             return "", &Error{Code: ErrI
1556         }
1557     }
1558     if p.flags&FoldCase == 0 {
1559         class = appendRange(class, lo, hi)
1560     } else {
1561         class = appendFoldedRange(class, lo,
1562     }
1563 }
1564 t = t[1:] // chop ]
1565
1566 // Use &re.Rune instead of &class to avoid allocatio
1567 re.Rune = class
1568 class = cleanClass(&re.Rune)
1569 if sign < 0 {
1570     class = negateClass(class)
1571 }

```

```

1572         re.Rune = class
1573         p.push(re)
1574         return t, nil
1575     }
1576
1577     // cleanClass sorts the ranges (pairs of elements of r),
1578     // merges them, and eliminates duplicates.
1579     func cleanClass(rp [][]rune) [][]rune {
1580
1581         // Sort by lo increasing, hi decreasing to break tie
1582         sort.Sort(ranges{rp})
1583
1584         r := *rp
1585         if len(r) < 2 {
1586             return r
1587         }
1588
1589         // Merge abutting, overlapping.
1590         w := 2 // write index
1591         for i := 2; i < len(r); i += 2 {
1592             lo, hi := r[i], r[i+1]
1593             if lo <= r[w-1]+1 {
1594                 // merge with previous range
1595                 if hi > r[w-1] {
1596                     r[w-1] = hi
1597                 }
1598                 continue
1599             }
1600             // new disjoint range
1601             r[w] = lo
1602             r[w+1] = hi
1603             w += 2
1604         }
1605
1606         return r[:w]
1607     }
1608
1609     // appendLiteral returns the result of appending the literal
1610     func appendLiteral(r []rune, x rune, flags Flags) []rune {
1611         if flags&FoldCase != 0 {
1612             return appendFoldedRange(r, x, x)
1613         }
1614         return appendRange(r, x, x)
1615     }
1616
1617     // appendRange returns the result of appending the range lo-
1618     func appendRange(r []rune, lo, hi rune) []rune {
1619         // Expand last range or next to last range if it ove
1620         // Checking two ranges helps when appending case-fo

```

```

1621 // alphabets, so that one range can be expanding A-Z
1622 // other expanding a-z.
1623 n := len(r)
1624 for i := 2; i <= 4; i += 2 { // twice, using i=2, i=
1625     if n >= i {
1626         rlo, rhi := r[n-i], r[n-i+1]
1627         if lo <= rhi+1 && rlo <= hi+1 {
1628             if lo < rlo {
1629                 r[n-i] = lo
1630             }
1631             if hi > rhi {
1632                 r[n-i+1] = hi
1633             }
1634             return r
1635         }
1636     }
1637 }
1638
1639 return append(r, lo, hi)
1640 }
1641
1642 const (
1643     // minimum and maximum runes involved in folding.
1644     // checked during test.
1645     minFold = 0x0041
1646     maxFold = 0x1044f
1647 )
1648
1649 // appendFoldedRange returns the result of appending the ran
1650 // and its case folding-equivalent runes to the class r.
1651 func appendFoldedRange(r []rune, lo, hi rune) []rune {
1652     // Optimizations.
1653     if lo <= minFold && hi >= maxFold {
1654         // Range is full: folding can't add more.
1655         return appendRange(r, lo, hi)
1656     }
1657     if hi < minFold || lo > maxFold {
1658         // Range is outside folding possibilities.
1659         return appendRange(r, lo, hi)
1660     }
1661     if lo < minFold {
1662         // [lo, minFold-1] needs no folding.
1663         r = appendRange(r, lo, minFold-1)
1664         lo = minFold
1665     }
1666     if hi > maxFold {
1667         // [maxFold+1, hi] needs no folding.
1668         r = appendRange(r, maxFold+1, hi)
1669         hi = maxFold

```

```

1670     }
1671
1672     // Brute force. Depend on appendRange to coalesce r
1673     for c := lo; c <= hi; c++ {
1674         r = appendRange(r, c, c)
1675         f := unicode.SimpleFold(c)
1676         for f != c {
1677             r = appendRange(r, f, f)
1678             f = unicode.SimpleFold(f)
1679         }
1680     }
1681     return r
1682 }
1683
1684 // appendClass returns the result of appending the class x t
1685 // It assume x is clean.
1686 func appendClass(r []rune, x []rune) []rune {
1687     for i := 0; i < len(x); i += 2 {
1688         r = appendRange(r, x[i], x[i+1])
1689     }
1690     return r
1691 }
1692
1693 // appendFolded returns the result of appending the case fol
1694 func appendFoldedClass(r []rune, x []rune) []rune {
1695     for i := 0; i < len(x); i += 2 {
1696         r = appendFoldedRange(r, x[i], x[i+1])
1697     }
1698     return r
1699 }
1700
1701 // appendNegatedClass returns the result of appending the ne
1702 // It assumes x is clean.
1703 func appendNegatedClass(r []rune, x []rune) []rune {
1704     nextLo := '\u0000'
1705     for i := 0; i < len(x); i += 2 {
1706         lo, hi := x[i], x[i+1]
1707         if nextLo <= lo-1 {
1708             r = appendRange(r, nextLo, lo-1)
1709         }
1710         nextLo = hi + 1
1711     }
1712     if nextLo <= unicode.MaxRune {
1713         r = appendRange(r, nextLo, unicode.MaxRune)
1714     }
1715     return r
1716 }
1717
1718 // appendTable returns the result of appending x to the clas
1719 func appendTable(r []rune, x *unicode.RangeTable) []rune {

```

```

1720     for _, xr := range x.R16 {
1721         lo, hi, stride := rune(xr.Lo), rune(xr.Hi),
1722         if stride == 1 {
1723             r = appendRange(r, lo, hi)
1724             continue
1725         }
1726         for c := lo; c <= hi; c += stride {
1727             r = appendRange(r, c, c)
1728         }
1729     }
1730     for _, xr := range x.R32 {
1731         lo, hi, stride := rune(xr.Lo), rune(xr.Hi),
1732         if stride == 1 {
1733             r = appendRange(r, lo, hi)
1734             continue
1735         }
1736         for c := lo; c <= hi; c += stride {
1737             r = appendRange(r, c, c)
1738         }
1739     }
1740     return r
1741 }
1742
1743 // appendNegatedTable returns the result of appending the ne
1744 func appendNegatedTable(r []rune, x *unicode.RangeTable) []r
1745     nextLo := '\u0000' // lo end of next class to add
1746     for _, xr := range x.R16 {
1747         lo, hi, stride := rune(xr.Lo), rune(xr.Hi),
1748         if stride == 1 {
1749             if nextLo <= lo-1 {
1750                 r = appendRange(r, nextLo, 1
1751             }
1752             nextLo = hi + 1
1753             continue
1754         }
1755         for c := lo; c <= hi; c += stride {
1756             if nextLo <= c-1 {
1757                 r = appendRange(r, nextLo, c
1758             }
1759             nextLo = c + 1
1760         }
1761     }
1762     for _, xr := range x.R32 {
1763         lo, hi, stride := rune(xr.Lo), rune(xr.Hi),
1764         if stride == 1 {
1765             if nextLo <= lo-1 {
1766                 r = appendRange(r, nextLo, 1
1767             }
1768             nextLo = hi + 1

```

```

1769             continue
1770         }
1771         for c := lo; c <= hi; c += stride {
1772             if nextLo <= c-1 {
1773                 r = appendRange(r, nextLo, c
1774             }
1775             nextLo = c + 1
1776         }
1777     }
1778     if nextLo <= unicode.MaxRune {
1779         r = appendRange(r, nextLo, unicode.MaxRune)
1780     }
1781     return r
1782 }
1783
1784 // negateClass overwrites r and returns r's negation.
1785 // It assumes the class r is already clean.
1786 func negateClass(r []rune) []rune {
1787     nextLo := '\u0000' // lo end of next class to add
1788     w := 0             // write index
1789     for i := 0; i < len(r); i += 2 {
1790         lo, hi := r[i], r[i+1]
1791         if nextLo <= lo-1 {
1792             r[w] = nextLo
1793             r[w+1] = lo - 1
1794             w += 2
1795         }
1796         nextLo = hi + 1
1797     }
1798     r = r[:w]
1799     if nextLo <= unicode.MaxRune {
1800         // It's possible for the negation to have on
1801         // range - this one - than the original clas
1802         r = append(r, nextLo, unicode.MaxRune)
1803     }
1804     return r
1805 }
1806
1807 // ranges implements sort.Interface on a []rune.
1808 // The choice of receiver type definition is strange
1809 // but avoids an allocation since we already have
1810 // a *[]rune.
1811 type ranges struct {
1812     p *[]rune
1813 }
1814
1815 func (ra ranges) Less(i, j int) bool {
1816     p := *ra.p
1817     i *= 2

```

```

1818         j *= 2
1819         return p[i] < p[j] || p[i] == p[j] && p[i+1] > p[j+1]
1820     }
1821
1822     func (ra ranges) Len() int {
1823         return len(*ra.p) / 2
1824     }
1825
1826     func (ra ranges) Swap(i, j int) {
1827         p := *ra.p
1828         i *= 2
1829         j *= 2
1830         p[i], p[i+1], p[j], p[j+1] = p[j], p[j+1], p[i], p[i+1]
1831     }
1832
1833     func checkUTF8(s string) error {
1834         for s != "" {
1835             rune, size := utf8.DecodeRuneInString(s)
1836             if rune == utf8.RuneError && size == 1 {
1837                 return &Error{Code: ErrInvalidUTF8,
1838                     }
1839             }
1840             s = s[size:]
1841         }
1842         return nil
1843     }
1844
1845     func nextRune(s string) (c rune, t string, err error) {
1846         c, size := utf8.DecodeRuneInString(s)
1847         if c == utf8.RuneError && size == 1 {
1848             return 0, "", &Error{Code: ErrInvalidUTF8, E
1849         }
1850         return c, s[size:], nil
1851     }
1852
1853     func isalnum(c rune) bool {
1854         return '0' <= c && c <= '9' || 'A' <= c && c <= 'Z'
1855     }
1856
1857     func unhex(c rune) rune {
1858         if '0' <= c && c <= '9' {
1859             return c - '0'
1860         }
1861         if 'a' <= c && c <= 'f' {
1862             return c - 'a' + 10
1863         }
1864         if 'A' <= c && c <= 'F' {
1865             return c - 'A' + 10
1866         }
1867         return -1
1868     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/regexp/syntax/perl_groups.go

```
1 // GENERATED BY make_perl_groups.pl; DO NOT EDIT.
2 // make_perl_groups.pl >perl_groups.go
3
4 package syntax
5
6 var code1 = []rune{ /* \d */
7     0x30, 0x39,
8 }
9
10 var code2 = []rune{ /* \s */
11     0x9, 0xa,
12     0xc, 0xd,
13     0x20, 0x20,
14 }
15
16 var code3 = []rune{ /* \w */
17     0x30, 0x39,
18     0x41, 0x5a,
19     0x5f, 0x5f,
20     0x61, 0x7a,
21 }
22
23 var perlGroup = map[string]charGroup{
24     `d`: {+1, code1},
25     `D`: {-1, code1},
26     `s`: {+1, code2},
27     `S`: {-1, code2},
28     `w`: {+1, code3},
29     `W`: {-1, code3},
30 }
31 var code4 = []rune{ /* [:alnum:] */
32     0x30, 0x39,
33     0x41, 0x5a,
34     0x61, 0x7a,
35 }
36
37 var code5 = []rune{ /* [:alpha:] */
38     0x41, 0x5a,
39     0x61, 0x7a,
40 }
41
```

```
42 var code6 = []rune{ /* [:ascii:] */
43     0x0, 0x7f,
44 }
45
46 var code7 = []rune{ /* [:blank:] */
47     0x9, 0x9,
48     0x20, 0x20,
49 }
50
51 var code8 = []rune{ /* [:cntrl:] */
52     0x0, 0x1f,
53     0x7f, 0x7f,
54 }
55
56 var code9 = []rune{ /* [:digit:] */
57     0x30, 0x39,
58 }
59
60 var code10 = []rune{ /* [:graph:] */
61     0x21, 0x7e,
62 }
63
64 var code11 = []rune{ /* [:lower:] */
65     0x61, 0x7a,
66 }
67
68 var code12 = []rune{ /* [:print:] */
69     0x20, 0x7e,
70 }
71
72 var code13 = []rune{ /* [:punct:] */
73     0x21, 0x2f,
74     0x3a, 0x40,
75     0x5b, 0x60,
76     0x7b, 0x7e,
77 }
78
79 var code14 = []rune{ /* [:space:] */
80     0x9, 0xd,
81     0x20, 0x20,
82 }
83
84 var code15 = []rune{ /* [:upper:] */
85     0x41, 0x5a,
86 }
87
88 var code16 = []rune{ /* [:word:] */
89     0x30, 0x39,
90     0x41, 0x5a,
91     0x5f, 0x5f,
```

```

92         0x61, 0x7a,
93     }
94
95     var code17 = []rune{ /* [:xdigit:] */
96         0x30, 0x39,
97         0x41, 0x46,
98         0x61, 0x66,
99     }
100
101     var posixGroup = map[string]charGroup{
102         `[:alnum:]`:    {+1, code4},
103         `[:^alnum:]`:   {-1, code4},
104         `[:alpha:]`:    {+1, code5},
105         `[:^alpha:]`:   {-1, code5},
106         `[:ascii:]`:    {+1, code6},
107         `[:^ascii:]`:   {-1, code6},
108         `[:blank:]`:    {+1, code7},
109         `[:^blank:]`:   {-1, code7},
110         `[:cntrl:]`:    {+1, code8},
111         `[:^cntrl:]`:   {-1, code8},
112         `[:digit:]`:    {+1, code9},
113         `[:^digit:]`:   {-1, code9},
114         `[:graph:]`:    {+1, code10},
115         `[:^graph:]`:   {-1, code10},
116         `[:lower:]`:    {+1, code11},
117         `[:^lower:]`:   {-1, code11},
118         `[:print:]`:    {+1, code12},
119         `[:^print:]`:   {-1, code12},
120         `[:punct:]`:    {+1, code13},
121         `[:^punct:]`:   {-1, code13},
122         `[:space:]`:    {+1, code14},
123         `[:^space:]`:   {-1, code14},
124         `[:upper:]`:    {+1, code15},
125         `[:^upper:]`:   {-1, code15},
126         `[:word:]`:     {+1, code16},
127         `[:^word:]`:    {-1, code16},
128         `[:xdigit:]`:   {+1, code17},
129         `[:^xdigit:]`: {-1, code17},
130     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/regexp/syntax/prog.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package syntax
6
7 import (
8     "bytes"
9     "strconv"
10    "unicode"
11 )
12
13 // Compiled program.
14 // May not belong in this package, but convenient for now.
15
16 // A Prog is a compiled regular expression program.
17 type Prog struct {
18     Inst []Inst
19     Start int // index of start instruction
20     NumCap int // number of InstCapture insts in re
21 }
22
23 // An InstOp is an instruction opcode.
24 type InstOp uint8
25
26 const (
27     InstAlt InstOp = iota
28     InstAltMatch
29     InstCapture
30     InstEmptyWidth
31     InstMatch
32     InstFail
33     InstNop
34     InstRune
35     InstRune1
36     InstRuneAny
37     InstRuneAnyNotNL
38 )
39
40 // An EmptyOp specifies a kind or mixture of zero-width asse
41 type EmptyOp uint8
```

```

42
43 const (
44     EmptyBeginLine EmptyOp = 1 << iota
45     EmptyEndLine
46     EmptyBeginText
47     EmptyEndText
48     EmptyWordBoundary
49     EmptyNoWordBoundary
50 )
51
52 // EmptyOpContext returns the zero-width assertions
53 // satisfied at the position between the runes r1 and r2.
54 // Passing r1 == -1 indicates that the position is
55 // at the beginning of the text.
56 // Passing r2 == -1 indicates that the position is
57 // at the end of the text.
58 func EmptyOpContext(r1, r2 rune) EmptyOp {
59     var op EmptyOp
60     if r1 < 0 {
61         op |= EmptyBeginText | EmptyBeginLine
62     }
63     if r1 == '\n' {
64         op |= EmptyBeginLine
65     }
66     if r2 < 0 {
67         op |= EmptyEndText | EmptyEndLine
68     }
69     if r2 == '\n' {
70         op |= EmptyEndLine
71     }
72     if IsWordChar(r1) != IsWordChar(r2) {
73         op |= EmptyWordBoundary
74     } else {
75         op |= EmptyNoWordBoundary
76     }
77     return op
78 }
79
80 // IsWordChar reports whether r is consider a ``word charact
81 // during the evaluation of the \b and \B zero-width asserti
82 // These assertions are ASCII-only: the word characters are
83 func IsWordChar(r rune) bool {
84     return 'A' <= r && r <= 'Z' || 'a' <= r && r <= 'z'
85 }
86
87 // An Inst is a single instruction in a regular expression p
88 type Inst struct {
89     Op    InstOp
90     Out   uint32 // all but InstMatch, InstFail
91     Arg   uint32 // InstAlt, InstAltMatch, InstCapture, I

```

```

92         Rune []rune
93     }
94
95     func (p *Prog) String() string {
96         var b bytes.Buffer
97         dumpProg(&b, p)
98         return b.String()
99     }
100
101     // skipNop follows any no-op or capturing instructions
102     // and returns the resulting pc.
103     func (p *Prog) skipNop(pc uint32) *Inst {
104         i := &p.Inst[pc]
105         for i.Op == InstNop || i.Op == InstCapture {
106             pc = i.Out
107             i = &p.Inst[pc]
108         }
109         return i
110     }
111
112     // op returns i.Op but merges all the Rune special cases into
113     // a single InstOp.
114     func (i *Inst) op() InstOp {
115         op := i.Op
116         switch op {
117             case InstRune1, InstRuneAny, InstRuneAnyNotNL:
118                 op = InstRune
119         }
120         return op
121     }
122
123     // Prefix returns a literal string that all matches for the
124     // regexp must start with. Complete is true if the prefix
125     // is the entire match.
126     func (p *Prog) Prefix() (prefix string, complete bool) {
127         i := p.skipNop(uint32(p.Start))
128
129         // Avoid allocation of buffer if prefix is empty.
130         if i.op() != InstRune || len(i.Rune) != 1 {
131             return "", i.Op == InstMatch
132         }
133
134         // Have prefix; gather characters.
135         var buf bytes.Buffer
136         for i.op() == InstRune && len(i.Rune) == 1 && Flags(
137             buf.WriteRune(i.Rune[0])
138             i = p.skipNop(i.Out)
139         )
140         return buf.String(), i.Op == InstMatch

```

```

141
142 // StartCond returns the leading empty-width conditions that
143 // be true in any match. It returns ^EmptyOp(0) if no match
144 func (p *Prog) StartCond() EmptyOp {
145     var flag EmptyOp
146     pc := uint32(p.Start)
147     i := &p.Inst[pc]
148 Loop:
149     for {
150         switch i.Op {
151         case InstEmptyWidth:
152             flag |= EmptyOp(i.Arg)
153         case InstFail:
154             return ^EmptyOp(0)
155         case InstCapture, InstNop:
156             // skip
157         default:
158             break Loop
159         }
160         pc = i.Out
161         i = &p.Inst[pc]
162     }
163     return flag
164 }
165
166 // MatchRune returns true if the instruction matches (and co
167 // It should only be called when i.Op == InstRune.
168 func (i *Inst) MatchRune(r rune) bool {
169     rune := i.Rune
170
171     // Special case: single-rune slice is from literal s
172     if len(rune) == 1 {
173         r0 := rune[0]
174         if r == r0 {
175             return true
176         }
177         if Flags(i.Arg)&FoldCase != 0 {
178             for r1 := unicode.SimpleFold(r0); r1
179                 if r == r1 {
180                     return true
181                 }
182             }
183         }
184     return false
185 }
186
187 // Peek at the first few pairs.
188 // Should handle ASCII well.
189 for j := 0; j < len(rune) && j <= 8; j += 2 {

```

```

190         if r < rune[j] {
191             return false
192         }
193         if r <= rune[j+1] {
194             return true
195         }
196     }
197
198     // Otherwise binary search.
199     lo := 0
200     hi := len(rune) / 2
201     for lo < hi {
202         m := lo + (hi-lo)/2
203         if c := rune[2*m]; c <= r {
204             if r <= rune[2*m+1] {
205                 return true
206             }
207             lo = m + 1
208         } else {
209             hi = m
210         }
211     }
212     return false
213 }
214
215 // As per re2's Prog::IsWordChar. Determines whether rune is
216 // Since we act on runes, it would be easy to support Unico
217 func wordRune(r rune) bool {
218     return r == '_' ||
219         ('A' <= r && r <= 'Z') ||
220         ('a' <= r && r <= 'z') ||
221         ('0' <= r && r <= '9')
222 }
223
224 // MatchEmptyWidth returns true if the instruction matches
225 // an empty string between the runes before and after.
226 // It should only be called when i.Op == InstEmptyWidth.
227 func (i *Inst) MatchEmptyWidth(before rune, after rune) bool
228     switch EmptyOp(i.Arg) {
229     case EmptyBeginLine:
230         return before == '\n' || before == -1
231     case EmptyEndLine:
232         return after == '\n' || after == -1
233     case EmptyBeginText:
234         return before == -1
235     case EmptyEndText:
236         return after == -1
237     case EmptyWordBoundary:
238         return wordRune(before) != wordRune(after)
239     case EmptyNoWordBoundary:

```

```

240             return wordRune(before) == wordRune(after)
241         }
242         panic("unknown empty width arg")
243     }
244
245     func (i *Inst) String() string {
246         var b bytes.Buffer
247         dumpInst(&b, i)
248         return b.String()
249     }
250
251     func bw(b *bytes.Buffer, args ...string) {
252         for _, s := range args {
253             b.WriteString(s)
254         }
255     }
256
257     func dumpProg(b *bytes.Buffer, p *Prog) {
258         for j := range p.Inst {
259             i := &p.Inst[j]
260             pc := strconv.Itoa(j)
261             if len(pc) < 3 {
262                 b.WriteString("   "[len(pc):])
263             }
264             if j == p.Start {
265                 pc += "*"
266             }
267             bw(b, pc, "\t")
268             dumpInst(b, i)
269             bw(b, "\n")
270         }
271     }
272
273     func u32(i uint32) string {
274         return strconv.FormatUint(uint64(i), 10)
275     }
276
277     func dumpInst(b *bytes.Buffer, i *Inst) {
278         switch i.Op {
279         case InstAlt:
280             bw(b, "alt -> ", u32(i.Out), ", ", u32(i.Arg)
281         case InstAltMatch:
282             bw(b, "altmatch -> ", u32(i.Out), ", ", u32(
283         case InstCapture:
284             bw(b, "cap ", u32(i.Arg), " -> ", u32(i.Out)
285         case InstEmptyWidth:
286             bw(b, "empty ", u32(i.Arg), " -> ", u32(i.Ou
287         case InstMatch:
288             bw(b, "match")

```

```

289     case InstFail:
290         bw(b, "fail")
291     case InstNop:
292         bw(b, "nop -> ", u32(i.Out))
293     case InstRune:
294         if i.Rune == nil {
295             // shouldn't happen
296             bw(b, "rune <nil>")
297         }
298         bw(b, "rune ", strconv.QuoteToASCII(string(i
299             if Flags(i.Arg)&FoldCase != 0 {
300                 bw(b, "/i")
301             }
302             bw(b, " -> ", u32(i.Out))
303     case InstRune1:
304         bw(b, "rune1 ", strconv.QuoteToASCII(string(
305     case InstRuneAny:
306         bw(b, "any -> ", u32(i.Out))
307     case InstRuneAnyNotNL:
308         bw(b, "anynotnl -> ", u32(i.Out))
309     }
310 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/regexp/syntax/regexp.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package syntax
6
7 // Note to implementers:
8 // In this package, re is always a *Regexp and r is always a
9
10 import (
11     "bytes"
12     "strconv"
13     "strings"
14     "unicode"
15 )
16
17 // A Regexp is a node in a regular expression syntax tree.
18 type Regexp struct {
19     Op      Op // operator
20     Flags   Flags
21     Sub     []*Regexp // subexpressions, if any
22     Sub0    [1]*Regexp // storage for short Sub
23     Rune    []rune    // matched runes, for OpLiteral,
24     Rune0   [2]rune   // storage for short Rune
25     Min, Max int        // min, max for OpRepeat
26     Cap     int        // capturing index, for OpCapture
27     Name    string     // capturing name, for OpCapture
28 }
29
30 // An Op is a single regular expression operator.
31 type Op uint8
32
33 // Operators are listed in precedence order, tightest binding
34 // Character class operators are listed simplest to most complex
35 // (OpLiteral, OpCharClass, OpAnyCharNotNL, OpAnyChar).
36
37 const (
38     OpNoMatch      Op = 1 + iota // matches no strings
39     OpEmptyMatch  // matches empty string
40     OpLiteral      // matches Runes sequentially
41     OpCharClass    // matches Runes in a class
```

```

42         OpAnyCharNotNL           // matches any charac
43         OpAnyChar                 // matches any charac
44         OpBeginLine               // matches empty stri
45         OpEndLine                 // matches empty stri
46         OpBeginText               // matches empty stri
47         OpEndText                 // matches empty stri
48         OpWordBoundary            // matches word bound
49         OpNoWordBoundary          // matches word non-b
50         OpCapture                 // capturing subexpre
51         OpStar                    // matches Sub[0] zer
52         OpPlus                    // matches Sub[0] one
53         OpQuest                   // matches Sub[0] zer
54         OpRepeat                  // matches Sub[0] at
55         OpConcat                  // matches concatenat
56         OpAlternate                // matches alternatio
57     )
58
59     const opPseudo Op = 128 // where pseudo-ops start
60
61     // Equal returns true if x and y have identical structure.
62     func (x *Regexp) Equal(y *Regexp) bool {
63         if x == nil || y == nil {
64             return x == y
65         }
66         if x.Op != y.Op {
67             return false
68         }
69         switch x.Op {
70         case OpEndText:
71             // The parse flags remember whether this is
72             if x.Flags&WasDollar != y.Flags&WasDollar {
73                 return false
74             }
75
76         case OpLiteral, OpCharClass:
77             if len(x.Rune) != len(y.Rune) {
78                 return false
79             }
80             for i, r := range x.Rune {
81                 if r != y.Rune[i] {
82                     return false
83                 }
84             }
85
86         case OpAlternate, OpConcat:
87             if len(x.Sub) != len(y.Sub) {
88                 return false
89             }
90             for i, sub := range x.Sub {
91                 if !sub.Equal(y.Sub[i]) {

```

```

92             return false
93         }
94     }
95
96     case OpStar, OpPlus, OpQuest:
97         if x.Flags&NonGreedy != y.Flags&NonGreedy ||
98             return false
99     }
100
101     case OpRepeat:
102         if x.Flags&NonGreedy != y.Flags&NonGreedy ||
103             return false
104     }
105
106     case OpCapture:
107         if x.Cap != y.Cap || x.Name != y.Name || !x.
108             return false
109     }
110 }
111 return true
112 }
113
114 // writeRegexp writes the Perl syntax for the regular expres
115 func writeRegexp(b *bytes.Buffer, re *Regexp) {
116     switch re.Op {
117     default:
118         b.WriteString("<invalid op" + strconv.Itoa(i
119     case OpNoMatch:
120         b.WriteString(`^[^x00-\x{10FFFF}]`)
121     case OpEmptyMatch:
122         b.WriteString(`(?:)`)
123     case OpLiteral:
124         if re.Flags&FoldCase != 0 {
125             b.WriteString(`(?i)`)
126         }
127         for _, r := range re.Rune {
128             escape(b, r, false)
129         }
130         if re.Flags&FoldCase != 0 {
131             b.WriteString(``)
132         }
133     case OpCharClass:
134         if len(re.Rune)%2 != 0 {
135             b.WriteString(`[invalid char class]`
136             break
137         }
138         b.WriteRune('[')
139         if len(re.Rune) == 0 {
140             b.WriteString(`^\x00-\x{10FFFF}`)

```

```

141         } else if re.Rune[0] == 0 && re.Rune[len(re.
142             // Contains 0 and MaxRune. Probably
143             // Print the gaps.
144             b.WriteRune('^')
145             for i := 1; i < len(re.Rune)-1; i +=
146                 lo, hi := re.Rune[i]+1, re.R
147                 escape(b, lo, lo == '-')
148                 if lo != hi {
149                     b.WriteRune('-')
150                     escape(b, hi, hi ==
151                         }
152                 }
153             } else {
154                 for i := 0; i < len(re.Rune); i += 2
155                     lo, hi := re.Rune[i], re.Run
156                     escape(b, lo, lo == '-')
157                     if lo != hi {
158                         b.WriteRune('-')
159                         escape(b, hi, hi ==
160                             }
161                     }
162                 }
163                 b.WriteRune(']')
164             case OpAnyCharNotNL:
165                 b.WriteString(`(?-s:.)`)
166             case OpAnyChar:
167                 b.WriteString(`(?s:.)`)
168             case OpBeginLine:
169                 b.WriteRune('^')
170             case OpEndLine:
171                 b.WriteRune('$')
172             case OpBeginText:
173                 b.WriteString(`\A`)
174             case OpEndText:
175                 if re.Flags&WasDollar != 0 {
176                     b.WriteString(`(?-m:$)`)
177                 } else {
178                     b.WriteString(`\z`)
179                 }
180             case OpWordBoundary:
181                 b.WriteString(`\b`)
182             case OpNowordBoundary:
183                 b.WriteString(`\B`)
184             case OpCapture:
185                 if re.Name != "" {
186                     b.WriteString(`(?P<`)
187                     b.WriteString(re.Name)
188                     b.WriteRune('>')
189                 } else {

```

```

190         b.WriteRune('(')
191     }
192     if re.Sub[0].Op != OpEmptyMatch {
193         writeRegexp(b, re.Sub[0])
194     }
195     b.WriteRune(')')
196 case OpStar, OpPlus, OpQuest, OpRepeat:
197     if sub := re.Sub[0]; sub.Op > OpCapture || s
198         b.WriteString(`(?:`)
199         writeRegexp(b, sub)
200         b.WriteString(`)`)
201     } else {
202         writeRegexp(b, sub)
203     }
204     switch re.Op {
205     case OpStar:
206         b.WriteRune('*')
207     case OpPlus:
208         b.WriteRune('+')
209     case OpQuest:
210         b.WriteRune('?')
211     case OpRepeat:
212         b.WriteRune('{')
213         b.WriteString(strconv.Itoa(re.Min))
214         if re.Max != re.Min {
215             b.WriteRune(',')
216             if re.Max >= 0 {
217                 b.WriteString(strcon
218             }
219         }
220         b.WriteRune('}')
221     }
222     if re.Flags&NonGreedy != 0 {
223         b.WriteRune('?')
224     }
225 case OpConcat:
226     for _, sub := range re.Sub {
227         if sub.Op == OpAlternate {
228             b.WriteString(`(?:`)
229             writeRegexp(b, sub)
230             b.WriteString(`)`)
231         } else {
232             writeRegexp(b, sub)
233         }
234     }
235 case OpAlternate:
236     for i, sub := range re.Sub {
237         if i > 0 {
238             b.WriteRune('|')
239         }

```

```

240             writeRegexp(b, sub)
241         }
242     }
243 }
244
245 func (re *Regexp) String() string {
246     var b bytes.Buffer
247     writeRegexp(&b, re)
248     return b.String()
249 }
250
251 const meta = `\.+*?()|[]{}^$`
252
253 func escape(b *bytes.Buffer, r rune, force bool) {
254     if unicode.IsPrint(r) {
255         if strings.IndexRune(meta, r) >= 0 || force
256             b.WriteRune('\\')
257     }
258     b.WriteRune(r)
259     return
260 }
261
262 switch r {
263 case '\a':
264     b.WriteString(`\a`)
265 case '\f':
266     b.WriteString(`\f`)
267 case '\n':
268     b.WriteString(`\n`)
269 case '\r':
270     b.WriteString(`\r`)
271 case '\t':
272     b.WriteString(`\t`)
273 case '\v':
274     b.WriteString(`\v`)
275 default:
276     if r < 0x100 {
277         b.WriteString(`\x`)
278         s := strconv.FormatInt(int64(r), 16)
279         if len(s) == 1 {
280             b.WriteRune('0')
281         }
282         b.WriteString(s)
283         break
284     }
285     b.WriteString(`\x{`)
286     b.WriteString(strconv.FormatInt(int64(r), 16)
287     b.WriteString(`}`)
288 }

```

```

289 }
290
291 // MaxCap walks the regexp to find the maximum capture index
292 func (re *Regexp) MaxCap() int {
293     m := 0
294     if re.Op == OpCapture {
295         m = re.Cap
296     }
297     for _, sub := range re.Sub {
298         if n := sub.MaxCap(); m < n {
299             m = n
300         }
301     }
302     return m
303 }
304
305 // CapNames walks the regexp to find the names of capturing
306 func (re *Regexp) CapNames() []string {
307     names := make([]string, re.MaxCap()+1)
308     re.capNames(names)
309     return names
310 }
311
312 func (re *Regexp) capNames(names []string) {
313     if re.Op == OpCapture {
314         names[re.Cap] = re.Name
315     }
316     for _, sub := range re.Sub {
317         sub.capNames(names)
318     }
319 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/regexp/syntax/simplify.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package syntax
6
7 // Simplify returns a regexp equivalent to re but without co
8 // and with various other simplifications, such as rewriting
9 // The resulting regexp will execute correctly but its strin
10 // will not produce the same parse tree, because capturing p
11 // may have been duplicated or removed. For example, the si
12 // for /(x){1,2}/ is /(x)(x)?/ but both parentheses capture
13 // The returned regexp may share structure with or be the or
14 func (re *Regexp) Simplify() *Regexp {
15     if re == nil {
16         return nil
17     }
18     switch re.Op {
19     case OpCapture, OpConcat, OpAlternate:
20         // Simplify children, building new Regexp if
21         nre := re
22         for i, sub := range re.Sub {
23             nsub := sub.Simplify()
24             if nre == re && nsub != sub {
25                 // Start a copy.
26                 nre = new(Regexp)
27                 *nre = *re
28                 nre.Rune = nil
29                 nre.Sub = append(nre.Sub[:0]
30             }
31             if nre != re {
32                 nre.Sub = append(nre.Sub, ns
33             }
34         }
35         return nre
36
37     case OpStar, OpPlus, OpQuest:
38         sub := re.Sub[0].Simplify()
39         return simplify1(re.Op, re.Flags, sub, re)
40
41     case OpRepeat:
```

```

42 // Special special case: x{0} matches the em
43 // and doesn't even need to consider x.
44 if re.Min == 0 && re.Max == 0 {
45     return &Regexp{Op: OpEmptyMatch}
46 }
47
48 // The fun begins.
49 sub := re.Sub[0].Simplify()
50
51 // x{n,} means at least n matches of x.
52 if re.Max == -1 {
53     // Special case: x{0,} is x*.
54     if re.Min == 0 {
55         return simplify1(OpStar, re.
56     }
57
58     // Special case: x{1,} is x+.
59     if re.Min == 1 {
60         return simplify1(OpPlus, re.
61     }
62
63     // General case: x{4,} is xxxx+.
64     nre := &Regexp{Op: OpConcat}
65     nre.Sub = nre.Sub0[:0]
66     for i := 0; i < re.Min-1; i++ {
67         nre.Sub = append(nre.Sub, su
68     }
69     nre.Sub = append(nre.Sub, simplify1(
70     return nre
71 }
72
73 // Special case x{0} handled above.
74
75 // Special case: x{1} is just x.
76 if re.Min == 1 && re.Max == 1 {
77     return sub
78 }
79
80 // General case: x{n,m} means n copies of x
81 // The machine will do less work if we nest
82 // so that x{2,5} = xx(x(x(x)?))?
83
84 // Build leading prefix: xx.
85 var prefix *Regexp
86 if re.Min > 0 {
87     prefix = &Regexp{Op: OpConcat}
88     prefix.Sub = prefix.Sub0[:0]
89     for i := 0; i < re.Min; i++ {
90         prefix.Sub = append(prefix.S
91     }

```

```

92         }
93
94         // Build and attach suffix: (x(x(x)?))?
95         if re.Max > re.Min {
96             suffix := simplify1(OpQuest, re.Flag
97                 for i := re.Min + 1; i < re.Max; i++
98                     nre2 := &Regexp{Op: OpConcat
99                         nre2.Sub = append(nre2.Sub0[
100                             suffix = simplify1(OpQuest,
101                                 }
102                                 if prefix == nil {
103                                     return suffix
104                                 }
105                                 prefix.Sub = append(prefix.Sub, suff
106                             }
107                             if prefix != nil {
108                                 return prefix
109                             }
110
111                             // Some degenerate case like min > max or mi
112                             // Handle as impossible match.
113                             return &Regexp{Op: OpNoMatch}
114                         }
115
116                     return re
117                 }
118
119         // simplify1 implements Simplify for the unary OpStar,
120         // OpPlus, and OpQuest operators. It returns the simple reg
121         // equivalent to
122         //
123         //     Regexp{Op: op, Flags: flags, Sub: {sub}}
124         //
125         // under the assumption that sub is already simple, and
126         // without first allocating that structure. If the regexp
127         // to be returned turns out to be equivalent to re, simplify
128         // returns re instead.
129         //
130         // simplify1 is factored out of Simplify because the impleme
131         // for other operators generates these unary expressions.
132         // Letting them call simplify1 makes sure the expressions th
133         // generate are simple.
134         func simplify1(op Op, flags Flags, sub, re *Regexp) *Regexp
135             // Special case: repeat the empty string as much as
136             // you want, but it's still the empty string.
137             if sub.Op == OpEmptyMatch {
138                 return sub
139             }
140             // The operators are idempotent if the flags match.

```

```
141         if op == sub.Op && flags&NonGreedy == sub.Flags&NonG
142             return sub
143     }
144     if re != nil && re.Op == op && re.Flags&NonGreedy ==
145         return re
146     }
147
148     re = &Regexp{Op: op, Flags: flags}
149     re.Sub = append(re.Sub0[:0], sub)
150     return re
151 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/runtime/compiler.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package runtime
6
7 // Compiler is the name of the compiler toolchain that built
8 // running binary. Known toolchains are:
9 //
10 //      gc      The 5g/6g/8g compiler suite at code.google.c
11 //      gccgo   The gccgo front end, part of the GCC compile
12 //
13 const Compiler = "gc"
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/runtime/debug.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package runtime
6
7 // Breakpoint() executes a breakpoint trap.
8 func Breakpoint()
9
10 // LockOSThread wires the calling goroutine to its current o
11 // Until the calling goroutine exits or calls UnlockOSThread
12 // execute in that thread, and no other goroutine can.
13 func LockOSThread()
14
15 // UnlockOSThread unwires the calling goroutine from its fix
16 // If the calling goroutine has not called LockOSThread, Unl
17 func UnlockOSThread()
18
19 // GOMAXPROCS sets the maximum number of CPUs that can be ex
20 // simultaneously and returns the previous setting. If n <
21 // change the current setting.
22 // The number of logical CPUs on the local machine can be qu
23 // This call will go away when the scheduler improves.
24 func GOMAXPROCS(n int) int
25
26 // NumCPU returns the number of logical CPUs on the local ma
27 func NumCPU() int
28
29 // NumCgoCall returns the number of cgo calls made by the cu
30 func NumCgoCall() int64
31
32 // NumGoroutine returns the number of goroutines that curren
33 func NumGoroutine() int
34
35 // MemProfileRate controls the fraction of memory allocation
36 // that are recorded and reported in the memory profile.
37 // The profiler aims to sample an average of
38 // one allocation per MemProfileRate bytes allocated.
39 //
40 // To include every allocated block in the profile, set MemP
41 // To turn off profiling entirely, set MemProfileRate to 0.
42 //
43 // The tools that process the memory profiles assume that th
44 // profile rate is constant across the lifetime of the progr
```

```

45 // and equal to the current value. Programs that change the
46 // memory profiling rate should do so just once, as early as
47 // possible in the execution of the program (for example,
48 // at the beginning of main).
49 var MemProfileRate int = 512 * 1024
50
51 // A MemProfileRecord describes the live objects allocated
52 // by a particular call sequence (stack trace).
53 type MemProfileRecord struct {
54     AllocBytes, FreeBytes      int64      // number of b
55     AllocObjects, FreeObjects int64      // number of o
56     Stack0                    [32]uintptr // stack trace
57 }
58
59 // InUseBytes returns the number of bytes in use (AllocBytes
60 func (r *MemProfileRecord) InUseBytes() int64 { return r.All
61
62 // InUseObjects returns the number of objects in use (AllocO
63 func (r *MemProfileRecord) InUseObjects() int64 {
64     return r.AllocObjects - r.FreeObjects
65 }
66
67 // Stack returns the stack trace associated with the record,
68 // a prefix of r.Stack0.
69 func (r *MemProfileRecord) Stack() []uintptr {
70     for i, v := range r.Stack0 {
71         if v == 0 {
72             return r.Stack0[0:i]
73         }
74     }
75     return r.Stack0[0:]
76 }
77
78 // MemProfile returns n, the number of records in the curren
79 // If len(p) >= n, MemProfile copies the profile into p and
80 // If len(p) < n, MemProfile does not change p and returns n
81 //
82 // If inuseZero is true, the profile includes allocation rec
83 // where r.AllocBytes > 0 but r.AllocBytes == r.FreeBytes.
84 // These are sites where memory was allocated, but it has al
85 // been released back to the runtime.
86 //
87 // Most clients should use the runtime/pprof package or
88 // the testing package's -test.memprofile flag instead
89 // of calling MemProfile directly.
90 func MemProfile(p []MemProfileRecord, inuseZero bool) (n int
91
92 // A StackRecord describes a single execution stack.
93 type StackRecord struct {
94     Stack0 [32]uintptr // stack trace for this record; e

```

```

95 }
96
97 // Stack returns the stack trace associated with the record,
98 // a prefix of r.Stack0.
99 func (r *StackRecord) Stack() []uintptr {
100     for i, v := range r.Stack0 {
101         if v == 0 {
102             return r.Stack0[0:i]
103         }
104     }
105     return r.Stack0[0:]
106 }
107
108 // ThreadCreateProfile returns n, the number of records in t
109 // If len(p) >= n, ThreadCreateProfile copies the profile in
110 // If len(p) < n, ThreadCreateProfile does not change p and
111 //
112 // Most clients should use the runtime/pprof package instead
113 // of calling ThreadCreateProfile directly.
114 func ThreadCreateProfile(p []StackRecord) (n int, ok bool)
115
116 // GoroutineProfile returns n, the number of records in the
117 // If len(p) >= n, GoroutineProfile copies the profile into
118 // If len(p) < n, GoroutineProfile does not change p and ret
119 //
120 // Most clients should use the runtime/pprof package instead
121 // of calling GoroutineProfile directly.
122 func GoroutineProfile(p []StackRecord) (n int, ok bool)
123
124 // CPUProfile returns the next chunk of binary CPU profiling
125 // blocking until data is available. If profiling is turned
126 // data accumulated while it was on has been returned, CPUPr
127 // The caller must save the returned data before calling CPU
128 // Most clients should use the runtime/pprof package or
129 // the testing package's -test.cpuprofile flag instead of ca
130 // CPUProfile directly.
131 func CPUProfile() []byte
132
133 // SetCPUProfileRate sets the CPU profiling rate to hz sampl
134 // If hz <= 0, SetCPUProfileRate turns off profiling.
135 // If the profiler is on, the rate cannot be changed without
136 // Most clients should use the runtime/pprof package or
137 // the testing package's -test.cpuprofile flag instead of ca
138 // SetCPUProfileRate directly.
139 func SetCPUProfileRate(hz int)
140
141 // Stack formats a stack trace of the calling goroutine into
142 // and returns the number of bytes written to buf.
143 // If all is true, Stack formats stack traces of all other g

```

```
144 // into buf after the trace for the current goroutine.  
145 func Stack(buf []byte, all bool) int
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/runtime/error.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package runtime
6
7 // The Error interface identifies a run time error.
8 type Error interface {
9     error
10
11     // RuntimeError is a no-op function but
12     // serves to distinguish types that are runtime
13     // errors from ordinary errors: a type is a
14     // runtime error if it has a RuntimeError method.
15     RuntimeError()
16 }
17
18 // A TypeAssertionError explains a failed type assertion.
19 type TypeAssertionError struct {
20     interfaceString string
21     concreteString  string
22     assertedString  string
23     missingMethod   string // one method needed by Inter
24 }
25
26 func (*TypeAssertionError) RuntimeError() {}
27
28 func (e *TypeAssertionError) Error() string {
29     inter := e.interfaceString
30     if inter == "" {
31         inter = "interface"
32     }
33     if e.concreteString == "" {
34         return "interface conversion: " + inter + "
35     }
36     if e.missingMethod == "" {
37         return "interface conversion: " + inter + "
38         ", not " + e.assertedString
39     }
40     return "interface conversion: " + e.concreteString +
41         ": missing method " + e.missingMethod
42 }
43
44 // For calling from C.
```

```

45 func newTypeAssertionError(ps1, ps2, ps3 *string, pmeth *str
46     var s1, s2, s3, meth string
47
48     if ps1 != nil {
49         s1 = *ps1
50     }
51     if ps2 != nil {
52         s2 = *ps2
53     }
54     if ps3 != nil {
55         s3 = *ps3
56     }
57     if pmeth != nil {
58         meth = *pmeth
59     }
60     *ret = &TypeAssertionError{s1, s2, s3, meth}
61 }
62
63 // An errorString represents a runtime error described by a
64 type errorString string
65
66 func (e errorString) RuntimeError() {}
67
68 func (e errorString) Error() string {
69     return "runtime error: " + string(e)
70 }
71
72 // For calling from C.
73 func newErrorString(s string, ret *interface{}) {
74     *ret = errorString(s)
75 }
76
77 type stringer interface {
78     String() string
79 }
80
81 func typestring(interface{}) string
82
83 // For calling from C.
84 // Prints an argument passed to panic.
85 // There's room for arbitrary complexity here, but we keep i
86 // simple and handle just a few important cases: int, string
87 func printany(i interface{}) {
88     switch v := i.(type) {
89     case nil:
90         print("nil")
91     case stringer:
92         print(v.String())
93     case error:
94         print(v.Error())

```

```
95         case int:
96             print(v)
97         case string:
98             print(v)
99         default:
100            print("(", typestring(i), ") ", i)
101     }
102 }
103
104 // called from generated code
105 func panicwrap(pkg, typ, meth string) {
106     panic("value method " + pkg + "." + typ + "." + meth
107 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/runtime/extern.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6     Package runtime contains operations that interact wi
7     such as functions to control goroutines. It also inc
8     used by the reflect package; see reflect's documenta
9     interface to the run-time type system.
10 */
11 package runtime
12
13 // Gosched yields the processor, allowing other goroutines t
14 // suspend the current goroutine, so execution resumes autom
15 func Gosched()
16
17 // Goexit terminates the goroutine that calls it. No other
18 // Goexit runs all deferred calls before terminating the gor
19 func Goexit()
20
21 // Caller reports file and line number information about fun
22 // the calling goroutine's stack. The argument skip is the
23 // to ascend, with 1 identifying the caller of Caller. (For
24 // meaning of skip differs between Caller and Callers.) The
25 // program counter, file name, and line number within the fi
26 // call. The boolean ok is false if it was not possible to
27 func Caller(skip int) (pc uintptr, file string, line int, ok
28
29 // Callers fills the slice pc with the program counters of f
30 // on the calling goroutine's stack. The argument skip is t
31 // to skip before recording in pc, with 0 starting at the ca
32 // It returns the number of entries written to pc.
33 func Callers(skip int, pc []uintptr) int
34
35 type Func struct { // Keep in sync with runtime.h:struct Fun
36     name    string
37     typ     string // go type string
38     src     string // src file name
39     pcln   []byte // pc/ln tab for this func
40     entry  uintptr // entry pc
41     pc0    uintptr // starting pc, ln for table
42     ln0    int32
43     frame  int32 // stack frame size
44     args   int32 // number of 32-bit in/out args
```

```

45         locals int32 // number of 32-bit locals
46     }
47
48     // FuncForPC returns a *Func describing the function that co
49     // given program counter address, or else nil.
50     func FuncForPC(pc uintptr) *Func
51
52     // Name returns the name of the function.
53     func (f *Func) Name() string { return f.name }
54
55     // Entry returns the entry address of the function.
56     func (f *Func) Entry() uintptr { return f.entry }
57
58     // FileLine returns the file name and line number of the
59     // source code corresponding to the program counter pc.
60     // The result will not be accurate if pc is not a program
61     // counter within f.
62     func (f *Func) FileLine(pc uintptr) (file string, line int)
63         return funcline_go(f, pc)
64     }
65
66     // implemented in symtab.c
67     func funcline_go(*Func, uintptr) (string, int)
68
69     // mid returns the current os thread (m) id.
70     func mid() uint32
71
72     // SetFinalizer sets the finalizer associated with x to f.
73     // When the garbage collector finds an unreachable block
74     // with an associated finalizer, it clears the association a
75     // f(x) in a separate goroutine. This makes x reachable aga
76     // now without an associated finalizer. Assuming that SetFi
77     // is not called again, the next time the garbage collector
78     // that x is unreachable, it will free x.
79     //
80     // SetFinalizer(x, nil) clears any finalizer associated with
81     //
82     // The argument x must be a pointer to an object allocated b
83     // calling new or by taking the address of a composite liter
84     // The argument f must be a function that takes a single arg
85     // of x's type and can have arbitrary ignored return values.
86     // If either of these is not true, SetFinalizer aborts the p
87     //
88     // Finalizers are run in dependency order: if A points at B,
89     // finalizers, and they are otherwise unreachable, only the
90     // for A runs; once A is freed, the finalizer for B can run.
91     // If a cyclic structure includes a block with a finalizer,
92     // cycle is not guaranteed to be garbage collected and the f
93     // is not guaranteed to run, because there is no ordering th
94     // respects the dependencies.

```

```

95 //
96 // The finalizer for x is scheduled to run at some arbitrary
97 // x becomes unreachable.
98 // There is no guarantee that finalizers will run before a p
99 // so typically they are useful only for releasing non-memor
100 // associated with an object during a long-running program.
101 // For example, an os.File object could use a finalizer to c
102 // associated operating system file descriptor when a progra
103 // an os.File without calling Close, but it would be a mista
104 // to depend on a finalizer to flush an in-memory I/O buffer
105 // bufio.Writer, because the buffer would not be flushed at
106 //
107 // A single goroutine runs all finalizers for a program, seq
108 // If a finalizer must run for a long time, it should do so
109 // a new goroutine.
110 func SetFinalizer(x, f interface{})
111
112 func getgoroot() string
113
114 // GOROOT returns the root of the Go tree.
115 // It uses the GOROOT environment variable, if set,
116 // or else the root used during the Go build.
117 func GOROOT() string {
118     s := getgoroot()
119     if s != "" {
120         return s
121     }
122     return defaultGoroot
123 }
124
125 // Version returns the Go tree's version string.
126 // It is either a sequence number or, when possible,
127 // a release tag like "release.2010-03-04".
128 // A trailing + indicates that the tree had local modificati
129 // at the time of the build.
130 func Version() string {
131     return theVersion
132 }
133
134 // GOOS is the running program's operating system target:
135 // one of darwin, freebsd, linux, and so on.
136 const GOOS string = theGoos
137
138 // GOARCH is the running program's architecture target:
139 // 386, amd64, or arm.
140 const GOARCH string = theGoarch

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/runtime/mem.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package runtime
6
7 import "unsafe"
8
9 // A MemStats records statistics about the memory allocator.
10 type MemStats struct {
11     // General statistics.
12     Alloc      uint64 // bytes allocated and still in us
13     TotalAlloc uint64 // bytes allocated (even if freed)
14     Sys        uint64 // bytes obtained from system (sho
15     Lookups    uint64 // number of pointer lookups
16     Mallocs    uint64 // number of mallocs
17     Frees      uint64 // number of frees
18
19     // Main allocation heap statistics.
20     HeapAlloc   uint64 // bytes allocated and still in
21     HeapSys     uint64 // bytes obtained from system
22     HeapIdle    uint64 // bytes in idle spans
23     HeapInuse   uint64 // bytes in non-idle span
24     HeapReleased uint64 // bytes released to the OS
25     HeapObjects uint64 // total number of allocated obj
26
27     // Low-level fixed-size structure allocator statisti
28     //     Inuse is bytes used now.
29     //     Sys is bytes obtained from system.
30     StackInuse  uint64 // bootstrap stacks
31     StackSys    uint64
32     MSpanInuse  uint64 // mspan structures
33     MSpanSys    uint64
34     MCacheInuse uint64 // mcache structures
35     MCacheSys   uint64
36     BuckHashSys uint64 // profiling bucket hash table
37
38     // Garbage collector statistics.
39     NextGC      uint64 // next run in HeapAlloc time (b
40     LastGC      uint64 // last run in absolute time (ns
41     PauseTotalNs uint64
42     PauseNs     [256]uint64 // most recent GC pause tim
43     NumGC       uint32
44     EnableGC    bool
```

```

45     DebugGC      bool
46
47     // Per-size allocation statistics.
48     // 61 is NumSizeClasses in the C code.
49     BySize [61]struct {
50         Size      uint32
51         Mallocs  uint64
52         Frees    uint64
53     }
54 }
55
56 var sizeof_C_MStats uintptr // filled in by malloc.goc
57
58 var memStats MemStats
59
60 func init() {
61     if sizeof_C_MStats != unsafe.Sizeof(memStats) {
62         println(sizeof_C_MStats, unsafe.Sizeof(memSt
63         panic("MStats vs MemStatsType size mismatch"
64     }
65 }
66
67 // ReadMemStats populates m with memory allocator statistics
68 func ReadMemStats(m *MemStats)
69
70 // GC runs a garbage collection.
71 func GC()

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/runtime/softfloat64.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Software IEEE754 64-bit floating point.
6 // Only referred to (and thus linked in) by arm port
7 // and by tests in this directory.
8
9 package runtime
10
11 const (
12     mantbits64 uint = 52
13     expbits64  uint = 11
14     bias64      = -1<<(expbits64-1) + 1
15
16     nan64 uint64 = (1<<expbits64-1)<<mantbits64 + 1
17     inf64 uint64 = (1<<expbits64 - 1) << mantbits64
18     neg64 uint64 = 1 << (expbits64 + mantbits64)
19
20     mantbits32 uint = 23
21     expbits32  uint = 8
22     bias32      = -1<<(expbits32-1) + 1
23
24     nan32 uint32 = (1<<expbits32-1)<<mantbits32 + 1
25     inf32 uint32 = (1<<expbits32 - 1) << mantbits32
26     neg32 uint32 = 1 << (expbits32 + mantbits32)
27 )
28
29 func funpack64(f uint64) (sign, mant uint64, exp int, inf, n
30     sign = f & (1 << (mantbits64 + expbits64))
31     mant = f & (1<<mantbits64 - 1)
32     exp = int(f>>mantbits64) & (1<<expbits64 - 1)
33
34     switch exp {
35     case 1<<expbits64 - 1:
36         if mant != 0 {
37             nan = true
38             return
39         }
40         inf = true
41         return
```

```

42
43     case 0:
44         // denormalized
45         if mant != 0 {
46             exp += bias64 + 1
47             for mant < 1<<mantbits64 {
48                 mant <<= 1
49                 exp--
50             }
51         }
52
53     default:
54         // add implicit top bit
55         mant |= 1 << mantbits64
56         exp += bias64
57     }
58     return
59 }
60
61 func funpack32(f uint32) (sign, mant uint32, exp int, inf, n
62     sign = f & (1 << (mantbits32 + expbits32))
63     mant = f & (1<<mantbits32 - 1)
64     exp = int(f>>mantbits32) & (1<<expbits32 - 1)
65
66     switch exp {
67     case 1<<expbits32 - 1:
68         if mant != 0 {
69             nan = true
70             return
71         }
72         inf = true
73         return
74
75     case 0:
76         // denormalized
77         if mant != 0 {
78             exp += bias32 + 1
79             for mant < 1<<mantbits32 {
80                 mant <<= 1
81                 exp--
82             }
83         }
84
85     default:
86         // add implicit top bit
87         mant |= 1 << mantbits32
88         exp += bias32
89     }
90     return
91 }

```

```

92
93 func fpack64(sign, mant uint64, exp int, trunc uint64) uint6
94     mant0, exp0, trunc0 := mant, exp, trunc
95     if mant == 0 {
96         return sign
97     }
98     for mant < 1<<mantbits64 {
99         mant <<= 1
100        exp--
101    }
102    for mant >= 4<<mantbits64 {
103        trunc |= mant & 1
104        mant >>= 1
105        exp++
106    }
107    if mant >= 2<<mantbits64 {
108        if mant&1 != 0 && (trunc != 0 || mant&2 != 0
109            mant++
110            if mant >= 4<<mantbits64 {
111                mant >>= 1
112                exp++
113            }
114        }
115        mant >>= 1
116        exp++
117    }
118    if exp >= 1<<expbits64-1+bias64 {
119        return sign ^ inf64
120    }
121    if exp < bias64+1 {
122        if exp < bias64-int(mantbits64) {
123            return sign | 0
124        }
125        // repeat expecting denormal
126        mant, exp, trunc = mant0, exp0, trunc0
127        for exp < bias64 {
128            trunc |= mant & 1
129            mant >>= 1
130            exp++
131        }
132        if mant&1 != 0 && (trunc != 0 || mant&2 != 0
133            mant++
134        }
135        mant >>= 1
136        exp++
137        if mant < 1<<mantbits64 {
138            return sign | mant
139        }
140    }

```

```

141         return sign | uint64(exp-bias64)<<mantbits64 | mant&
142     }
143
144 func fpack32(sign, mant uint32, exp int, trunc uint32) uint32
145     mant0, exp0, trunc0 := mant, exp, trunc
146     if mant == 0 {
147         return sign
148     }
149     for mant < 1<<mantbits32 {
150         mant <<= 1
151         exp--
152     }
153     for mant >= 4<<mantbits32 {
154         trunc |= mant & 1
155         mant >>= 1
156         exp++
157     }
158     if mant >= 2<<mantbits32 {
159         if mant&1 != 0 && (trunc != 0 || mant&2 != 0
160             mant++
161             if mant >= 4<<mantbits32 {
162                 mant >>= 1
163                 exp++
164             }
165         }
166         mant >>= 1
167         exp++
168     }
169     if exp >= 1<<expbits32-1+bias32 {
170         return sign ^ inf32
171     }
172     if exp < bias32+1 {
173         if exp < bias32-int(mantbits32) {
174             return sign | 0
175         }
176         // repeat expecting denormal
177         mant, exp, trunc = mant0, exp0, trunc0
178         for exp < bias32 {
179             trunc |= mant & 1
180             mant >>= 1
181             exp++
182         }
183         if mant&1 != 0 && (trunc != 0 || mant&2 != 0
184             mant++
185         }
186         mant >>= 1
187         exp++
188         if mant < 1<<mantbits32 {
189             return sign | mant

```

```

190         }
191     }
192     return sign | uint32(exp-bias32)<<mantbits32 | mant&
193 }
194
195 func fadd64(f, g uint64) uint64 {
196     fs, fm, fe, fi, fn := funpack64(f)
197     gs, gm, ge, gi, gn := funpack64(g)
198
199     // Special cases.
200     switch {
201     case fn || gn: // NaN + x or x + NaN = NaN
202         return nan64
203
204     case fi && gi && fs != gs: // +Inf + -Inf or -Inf +
205         return nan64
206
207     case fi: // ±Inf + g = ±Inf
208         return f
209
210     case gi: // f + ±Inf = ±Inf
211         return g
212
213     case fm == 0 && gm == 0 && fs != 0 && gs != 0: // -0
214         return f
215
216     case fm == 0: // 0 + g = g but 0 + -0 = +0
217         if gm == 0 {
218             g ^= gs
219         }
220         return g
221
222     case gm == 0: // f + 0 = f
223         return f
224
225     }
226
227     if fe < ge || fe == ge && fm < gm {
228         f, g, fs, fm, fe, gs, gm, ge = g, f, gs, gm,
229     }
230
231     shift := uint(fe - ge)
232     fm <<= 2
233     gm <<= 2
234     trunc := gm & (1<<shift - 1)
235     gm >>= shift
236     if fs == gs {
237         fm += gm
238     } else {
239         fm -= gm

```

```

240         if trunc != 0 {
241             fm--
242         }
243     }
244     if fm == 0 {
245         fs = 0
246     }
247     return fpack64(fs, fm, fe-2, trunc)
248 }
249
250 func fsub64(f, g uint64) uint64 {
251     return fadd64(f, fneg64(g))
252 }
253
254 func fneg64(f uint64) uint64 {
255     return f ^ (1 << (mantbits64 + expbits64))
256 }
257
258 func fmul64(f, g uint64) uint64 {
259     fs, fm, fe, fi, fn := funpack64(f)
260     gs, gm, ge, gi, gn := funpack64(g)
261
262     // Special cases.
263     switch {
264     case fn || gn: // NaN * g or f * NaN = NaN
265         return nan64
266
267     case fi && gi: // Inf * Inf = Inf (with sign adjuste
268         return f ^ gs
269
270     case fi && gm == 0, fm == 0 && gi: // 0 * Inf = Inf
271         return nan64
272
273     case fm == 0: // 0 * x = 0 (with sign adjusted)
274         return f ^ gs
275
276     case gm == 0: // x * 0 = 0 (with sign adjusted)
277         return g ^ fs
278     }
279
280     // 53-bit * 53-bit = 107- or 108-bit
281     lo, hi := mullu(fm, gm)
282     shift := mantbits64 - 1
283     trunc := lo & (1<<shift - 1)
284     mant := hi<<(64-shift) | lo>>shift
285     return fpack64(fs^gs, mant, fe+ge-1, trunc)
286 }
287
288 func fdiv64(f, g uint64) uint64 {

```

```

289     fs, fm, fe, fi, fn := funpack64(f)
290     gs, gm, ge, gi, gn := funpack64(g)
291
292     // Special cases.
293     switch {
294     case fn || gn: // NaN / g = f / NaN = NaN
295         return nan64
296
297     case fi && gi: // ±Inf / ±Inf = NaN
298         return nan64
299
300     case !fi && !gi && fm == 0 && gm == 0: // 0 / 0 = Na
301         return nan64
302
303     case fi, !gi && gm == 0: // Inf / g = f / 0 = Inf
304         return fs ^ gs ^ inf64
305
306     case gi, fm == 0: // f / Inf = 0 / g = Inf
307         return fs ^ gs ^ 0
308     }
309     -, -, -, - = fi, fn, gi, gn
310
311     // 53-bit<<54 / 53-bit = 53- or 54-bit.
312     shift := mantbits64 + 2
313     q, r := divlu(fm>>(64-shift), fm<<shift, gm)
314     return fpack64(fs^gs, q, fe-ge-2, r)
315 }
316
317 func f64to32(f uint64) uint32 {
318     fs, fm, fe, fi, fn := funpack64(f)
319     if fn {
320         return nan32
321     }
322     fs32 := uint32(fs >> 32)
323     if fi {
324         return fs32 ^ inf32
325     }
326     const d = mantbits64 - mantbits32 - 1
327     return fpack32(fs32, uint32(fm>>d), fe-1, uint32(fm&
328 }
329
330 func f32to64(f uint32) uint64 {
331     const d = mantbits64 - mantbits32
332     fs, fm, fe, fi, fn := funpack32(f)
333     if fn {
334         return nan64
335     }
336     fs64 := uint64(fs) << 32
337     if fi {

```

```

338         return fs64 ^ inf64
339     }
340     return fpack64(fs64, uint64(fm)<<d, fe, 0)
341 }
342
343 func fcmp64(f, g uint64) (cmp int, isnan bool) {
344     fs, fm, _, fi, fn := funpack64(f)
345     gs, gm, _, gi, gn := funpack64(g)
346
347     switch {
348     case fn, gn: // flag NaN
349         return 0, true
350
351     case !fi && !gi && fm == 0 && gm == 0: // ±0 == ±0
352         return 0, false
353
354     case fs > gs: // f < 0, g > 0
355         return -1, false
356
357     case fs < gs: // f > 0, g < 0
358         return +1, false
359
360     // Same sign, not NaN.
361     // Can compare encodings directly now.
362     // Reverse for sign.
363     case fs == 0 && f < g, fs != 0 && f > g:
364         return -1, false
365
366     case fs == 0 && f > g, fs != 0 && f < g:
367         return +1, false
368     }
369
370     // f == g
371     return 0, false
372 }
373
374 func f64toint(f uint64) (val int64, ok bool) {
375     fs, fm, fe, fi, fn := funpack64(f)
376
377     switch {
378     case fi, fn: // NaN
379         return 0, false
380
381     case fe < -1: // f < 0.5
382         return 0, false
383
384     case fe > 63: // f >= 2^63
385         if fs != 0 && fm == 0 { // f == -2^63
386             return -1 << 63, true
387         }

```

```

388         if fs != 0 {
389             return 0, false
390         }
391         return 0, false
392     }
393
394     for fe > int(mantbits64) {
395         fe--
396         fm <<= 1
397     }
398     for fe < int(mantbits64) {
399         fe++
400         fm >>= 1
401     }
402     val = int64(fm)
403     if fs != 0 {
404         val = -val
405     }
406     return val, true
407 }
408
409 func fintto64(val int64) (f uint64) {
410     fs := uint64(val) & (1 << 63)
411     mant := uint64(val)
412     if fs != 0 {
413         mant = -mant
414     }
415     return fpack64(fs, mant, int(mantbits64), 0)
416 }
417
418 // 64x64 -> 128 multiply.
419 // adapted from hacker's delight.
420 func mullu(u, v uint64) (lo, hi uint64) {
421     const (
422         s      = 32
423         mask = 1<<s - 1
424     )
425     u0 := u & mask
426     u1 := u >> s
427     v0 := v & mask
428     v1 := v >> s
429     w0 := u0 * v0
430     t := u1*v0 + w0>>s
431     w1 := t & mask
432     w2 := t >> s
433     w1 += u0 * v1
434     return u * v, u1*v1 + w2 + w1>>s
435 }
436

```

```

437 // 128/64 -> 64 quotient, 64 remainder.
438 // adapted from hacker's delight
439 func divlu(u1, u0, v uint64) (q, r uint64) {
440     const b = 1 << 32
441
442     if u1 >= v {
443         return 1<<64 - 1, 1<<64 - 1
444     }
445
446     // s = nlz(v); v <= s
447     s := uint(0)
448     for v&(1<<63) == 0 {
449         s++
450         v <= 1
451     }
452
453     vn1 := v >> 32
454     vn0 := v & (1<<32 - 1)
455     un32 := u1<<s | u0>>(64-s)
456     un10 := u0 << s
457     un1 := un10 >> 32
458     un0 := un10 & (1<<32 - 1)
459     q1 := un32 / vn1
460     rhat := un32 - q1*vn1
461
462     again1:
463     if q1 >= b || q1*vn0 > b*rhat+un1 {
464         q1--
465         rhat += vn1
466         if rhat < b {
467             goto again1
468         }
469     }
470
471     un21 := un32*b + un1 - q1*v
472     q0 := un21 / vn1
473     rhat = un21 - q0*vn1
474
475     again2:
476     if q0 >= b || q0*vn0 > b*rhat+un0 {
477         q0--
478         rhat += vn1
479         if rhat < b {
480             goto again2
481         }
482     }
483
484     return q1*b + q0, (un21*b + un0 - q0*v) >> s
485 }

```

```
486
487 // callable from C
488
489 func fadd64c(f, g uint64, ret *uint64)           { *ret = f
490 func fsub64c(f, g uint64, ret *uint64)           { *ret = f
491 func fmul64c(f, g uint64, ret *uint64)           { *ret = f
492 func fdiv64c(f, g uint64, ret *uint64)           { *ret = f
493 func fneg64c(f uint64, ret *uint64)              { *ret = f
494 func f32to64c(f uint32, ret *uint64)             { *ret = f
495 func f64to32c(f uint64, ret *uint32)             { *ret = f
496 func fcmp64c(f, g uint64, ret *int, retnan *bool) { *ret, *r
497 func fintto64c(val int64, ret *uint64)           { *ret = f
498 func f64tointc(f uint64, ret *int64, retok *bool) { *ret, *r
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/runtime/type.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6  * Runtime type representation.
7  * This file exists only to provide types that 6l can turn i
8  * DWARF information for use by gdb. Nothing else uses thes
9  * They should match the same types in ../reflect/type.go.
10 * For comments see ../reflect/type.go.
11 */
12
13 package runtime
14
15 import "unsafe"
16
17 type commonType struct {
18     size      uintptr
19     hash      uint32
20     _         uint8
21     align     uint8
22     fieldAlign uint8
23     kind      uint8
24     alg       *uintptr
25     string    *string
26     *uncommonType
27     ptrToThis *interface{}
28 }
29
30 type _method struct {
31     name      *string
32     pkgPath   *string
33     mtyp     *interface{}
34     typ      *interface{}
35     ifn      unsafe.Pointer
36     tfn      unsafe.Pointer
37 }
38
39 type uncommonType struct {
40     name      *string
41     pkgPath   *string
42     methods  []_method
43 }
44
```

```
45 type _imethod struct {
46     name      *string
47     pkgPath   *string
48     typ       *interface{}
49 }
50
51 type interfaceType struct {
52     commonType
53     methods []_imethod
54 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/runtime/zgoarch_amd64.go

```
1 // auto generated by go tool dist
2
3 package runtime
4
5 const theGoarch = `amd64`
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/runtime/zgoos_linux.go

```
1 // auto generated by go tool dist
2
3 package runtime
4
5 const theGoos = `linux`
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/runtime/zruntime_defs_linux_

```
1 // auto generated by go tool dist
2
3 package runtime
4 import "unsafe"
5 var _ unsafe.Pointer
6
7 type lock struct {
8     // (union)      key      uint
9     waitm      *m
10 }
11
12 type note struct {
13     // (union)      key      uint
14     waitm      *m
15 }
16
17 type _string struct {
18     str      *uint8
19     len      int
20 }
21
22 type iface struct {
23     tab      *itab
24     data      unsafe.Pointer
25 }
26
27 type eface struct {
28     _type      *_type
29     data      unsafe.Pointer
30 }
31
32 type _complex64 struct {
33     real      float32
34     imag      float32
35 }
36
37 type _complex128 struct {
38     real      float64
39     imag      float64
40 }
41
```

```

42 type slice struct {
43     array *uint8
44     len    uint
45     cap    uint
46 }
47
48 type gobuf struct {
49     sp *uint8
50     pc *uint8
51     g  *g
52 }
53
54 type g struct {
55     stackguard *uint8
56     stackbase  *uint8
57     _defer    *_defer
58     _panic    *_panic
59     sched     gobuf
60     gcstack   *uint8
61     gcsp      *uint8
62     gcguard   *uint8
63     stack0    *uint8
64     entry     *uint8
65     alllink   *g
66     param     unsafe.Pointer
67     status    int16
68     goid      int
69     selgen    uint
70     waitreason *int8
71     schedlink  *g
72     readyonstop uint8
73     ispanic    uint8
74     m          *m
75     lockedm    *m
76     idlem      *m
77     sig        int
78     writenbuf  int
79     writebuf   *uint8
80     sigcode0   uint64
81     sigcode1   uint64
82     sigpc      uint64
83     gopc       uint64
84     end        [0]uint64
85 }
86
87 type m struct {
88     g0 *g
89     morepc func()
90     moreargp unsafe.Pointer
91     morebuf gobuf

```

```

92         moreframesize    uint
93         moreargsize     uint
94         cret             uint64
95         procid          uint64
96         gsignal         *g
97         tls             [8]uint
98         curg            *g
99         id              int
100        mallocing        int
101        gcing           int
102        locks          int
103        nomemprof       int
104        waitnextg       int
105        dying           int
106        profilehz       int
107        helpgc          int
108        fastrand         uint
109        ncgocall        uint64
110        havenextg       note
111        nextg           *g
112        alllink          *m
113        schedlink        *m
114        machport         uint
115        mcache           *mcache
116        stackalloc       *fixalloc
117        lockedg          *g
118        idleg            *g
119        createstack      [32]uint64
120        freglo           [16]uint
121        freghi           [16]uint
122        fflag            uint
123        nextwaitm        *m
124        waitsema         uint64
125        waitsemacount    uint
126        waitsemalock     uint
127        end              [0]uint64
128    }
129
130    type stktop struct {
131        stackguard        *uint8
132        stackbase         *uint8
133        gobuf             gobuf
134        argsize           uint
135        argp              *uint8
136        free              uint64
137        _panic            uint8
138    }
139
140    type sigtab struct {

```

```

141         flags    int
142         name     *int8
143     }
144
145     type _func struct {
146         name     string
147         _type   string
148         src     string
149         pcIn    []byte
150         entry   uint64
151         pc0     uint64
152         ln0     int
153         frame   int
154         args    int
155         locals  int
156     }
157
158     type wincall struct {
159         fn        func(unsafe.Pointer)
160         n        uint64
161         args     unsafe.Pointer
162         r1       uint64
163         r2       uint64
164         err      uint64
165     }
166
167     type timers struct {
168         lock
169         timerproc *g
170         sleeping  uint8
171         rescheduling uint8
172         waitnote  note
173         t        **timer
174         len     int
175         cap    int
176     }
177
178     type timer struct {
179         i        int
180         when    int64
181         period  int64
182         f        func(int64, eface)
183         arg     eface
184     }
185
186     type alg struct {
187         hash    func(*uint64, uint64, unsafe.Pointer)
188         equal   func(*uint8, uint64, unsafe.Pointer, unsafe.
189         print   func(uint64, unsafe.Pointer)

```

```

190         copy    func(uint64, unsafe.Pointer, unsafe.Pointer)
191     }
192
193     var algarray    [22]alg
194     type _defer struct {
195         siz        int
196         nofree     uint8
197         argp       *uint8
198         pc         *uint8
199         fn         *uint8
200         link       *_defer
201         args       [8]uint8
202     }
203
204     type _panic struct {
205         arg         eface
206         stackbase   *uint8
207         link        *_panic
208         recovered   uint8
209     }
210
211     var emptystring string
212     var allg          *g
213     var lastg        *g
214     var allm          *m
215     var gomaxprocs   int
216     var singleproc   uint8
217     var panicking    uint
218     var gcwaiting    int
219     var goos         *int8
220     var ncpu         int
221     var iscgo        uint8
222     var worldsema    uint
223     type timespec struct {
224         tv_sec    int64
225         tv_nsec   int64
226     }
227
228     type timeval struct {
229         tv_sec    int64
230         tv_usec   int64
231     }
232
233     type sigaction struct {
234         sa_handler unsafe.Pointer
235         sa_flags    uint64
236         sa_restorer unsafe.Pointer
237         sa_mask     uint64
238     }
239

```

```

240 type siginfo struct {
241     si_signo      int
242     si_errno      int
243     si_code int
244     pad_cgo_0     [4]uint8
245     _sifields     [112]uint8
246 }
247
248 type itimerval struct {
249     it_interval  timeval
250     it_value     timeval
251 }
252
253 type usigset struct {
254     __val [16]uint64
255 }
256
257 type fpxreg struct {
258     significand [4]uint16
259     exponent    uint16
260     padding [3]uint16
261 }
262
263 type xmmreg struct {
264     element [4]uint
265 }
266
267 type fpstate struct {
268     cwd    uint16
269     swd    uint16
270     ftw    uint16
271     fop    uint16
272     rip    uint64
273     rdp    uint64
274     mxcsr  uint
275     mxcr_mask  uint
276     _st      [8]fpxreg
277     _xmm     [16]xmmreg
278     padding [24]uint
279 }
280
281 type fpxreg1 struct {
282     significand [4]uint16
283     exponent    uint16
284     padding [3]uint16
285 }
286
287 type xmmreg1 struct {
288     element [4]uint

```

```

289 }
290
291 type fpstate1 struct {
292     cwd      uint16
293     swd      uint16
294     ftw      uint16
295     fop      uint16
296     rip      uint64
297     rdp      uint64
298     mxcsr    uint
299     mxcr_mask uint
300     _st      [8]fpxreg1
301     _xmm     [16]xmmreg1
302     padding  [24]uint
303 }
304
305 type fpreg1 struct {
306     significand [4]uint16
307     exponent    uint16
308 }
309
310 type sigaltstack struct {
311     ss_sp *uint8
312     ss_flags int
313     pad_cgo_0 [4]uint8
314     ss_size uint64
315 }
316
317 type mcontext struct {
318     gregs [23]int64
319     fpregs *fpstate
320     __reserved1 [8]uint64
321 }
322
323 type ucontext struct {
324     uc_flags uint64
325     uc_link *ucontext
326     uc_stack sigaltstack
327     uc_mcontext mcontext
328     uc_sigmask usigset
329     __fpregs_mem fpstate
330 }
331
332 type sigcontext struct {
333     r8 uint64
334     r9 uint64
335     r10 uint64
336     r11 uint64
337     r12 uint64

```

```

338         r13      uint64
339         r14      uint64
340         r15      uint64
341         rdi      uint64
342         rsi      uint64
343         rbp      uint64
344         rbx      uint64
345         rdx      uint64
346         rax      uint64
347         rcx      uint64
348         rsp      uint64
349         rip      uint64
350         eflags   uint64
351         cs       uint16
352         gs       uint16
353         fs       uint16
354         __pad0   uint16
355         err      uint64
356         trapno   uint64
357         oldmask  uint64
358         cr2      uint64
359         fpstate  *fpstate1
360         __reserved1 [8]uint64
361     }
362
363     type mlink struct {
364         next      *mlink
365     }
366
367     type fixalloc struct {
368         size      uint64
369         alloc     func(uint64) unsafe.Pointer
370         first     func(unsafe.Pointer, *uint8)
371         arg       unsafe.Pointer
372         list      *mlink
373         chunk     *uint8
374         nchunk    uint
375         inuse     uint64
376         sys       uint64
377     }
378
379     type _1_ struct {
380         size      uint
381         nmalloc   uint64
382         nfree     uint64
383     }
384
385     type mstats struct {
386         alloc     uint64
387         total_alloc uint64

```

```

388         sys      uint64
389         nlookup uint64
390         nmalloc uint64
391         nfree   uint64
392         heap_alloc uint64
393         heap_sys  uint64
394         heap_idle uint64
395         heap_inuse uint64
396         heap_released uint64
397         heap_objects uint64
398         stacks_inuse uint64
399         stacks_sys  uint64
400         mspan_inuse uint64
401         mspan_sys  uint64
402         mcache_inuse uint64
403         mcache_sys uint64
404         buckhash_sys uint64
405         next_gc uint64
406         last_gc uint64
407         pause_total_ns uint64
408         pause_ns [256]uint64
409         numgc uint
410         enablegc uint8
411         debuggc uint8
412         by_size [61]_1_
413     }
414
415     var memstats mstats
416     var class_to_size [61]int
417     var class_to_allocnpages [61]int
418     var class_to_transfercount [61]int
419     type mcachelist struct {
420         list *mlink
421         nlist uint
422         nlistmin uint
423     }
424
425     type _2_ struct {
426         nmalloc int64
427         nfree int64
428     }
429
430     type mcache struct {
431         list [61]mcachelist
432         size uint64
433         local_cachealloc int64
434         local_objects int64
435         local_alloc int64
436         local_total_alloc int64

```

```

437         local_nmalloc    int64
438         local_nfree      int64
439         local_nlookup    int64
440         next_sample      int
441         local_by_size    [61]_2_
442     }
443
444     type mspan struct {
445         next      *mspan
446         prev      *mspan
447         allnext   *mspan
448         start     uint64
449         npages    uint64
450         freelist   *mlink
451         ref        uint
452         sizeclass  uint
453         state      uint
454         unusedsince int64
455         npreleased uint64
456         limit      *uint8
457     }
458
459     type mcentral struct {
460         lock
461         sizeclass    int
462         nonempty     mspan
463         empty        mspan
464         nfree        int
465     }
466
467     type _3_ struct {
468         mcentral
469         // (union)      pad      [64]uint8
470     }
471
472     type mheap struct {
473         lock
474         free      [256]mspan
475         large     mspan
476         allspans  *mspan
477         _map      [4194304]*mspan
478         bitmap    *uint8
479         bitmap_mapped uint64
480         arena_start *uint8
481         arena_used  *uint8
482         arena_end  *uint8
483         central   [61]_3_
484         spanalloc  fixalloc
485         cachealloc fixalloc

```

```

486 }
487
488 var checking    int
489 type sigset struct {
490     mask    [2]uint
491 }
492
493 type rlimit struct {
494     rlim_cur    uint64
495     rlim_max    uint64
496 }
497
498 var m0    m
499 var g0    g
500 var debug    int
501 type sched struct {
502     lock
503     gfree    *g
504     goidgen int
505     ghead    *g
506     gtail    *g
507     gwait    int
508     gcount   int
509     grunning    int
510     mhead    *m
511     mwait    int
512     mcount   int
513     atomic   uint
514     profilehz    int
515     init     uint8
516     lockmain    uint8
517     stopped note
518 }
519
520 var mwakeupp    *m
521 var scvg        *g
522 var libcgo_thread_start func(unsafe.Pointer)
523 type cgothreadstart struct {
524     m        *m
525     g        *g
526     fn        func()
527 }
528
529 type _4_ struct {
530     lock
531     fn        func(*uint64, int)
532     hz        int
533     pcbuf    [100]uint64
534 }
535

```

```

536 var prof          _4_
537 var libcgo_setenv func(**uint8)
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558 type commontype struct {
559     size      uint64
560     hash      uint
561     _unused   uint8
562     align     uint8
563     fieldalign      uint8
564     kind      uint8
565     alg       *alg
566     _string   *string
567     x         *uncommontype
568     ptrto    *_type
569 }
570
571 type method struct {
572     name      *string
573     pkgpath   *string
574     mtyp     *_type
575     typ      *_type
576     ifn      func()
577     tfn      func()
578 }
579
580 type uncommontype struct {
581     name      *string
582     pkgpath   *string
583     mhdr      []byte
584     m         [0]method

```

```

585 }
586
587 type _type struct {
588     _type    unsafe.Pointer
589     ptr      unsafe.Pointer
590     commontype
591 }
592
593 type imethod struct {
594     name      *string
595     pkgpath   *string
596     _type     *_type
597 }
598
599 type interfacetype struct {
600     _type
601     mhdr      []byte
602     m         [0]imethod
603 }
604
605 type maptype struct {
606     _type
607     key       *_type
608     elem      *_type
609 }
610
611 type chantype struct {
612     _type
613     elem      *_type
614     dir       uint64
615 }
616
617 type slicetype struct {
618     _type
619     elem      *_type
620 }
621
622 type functype struct {
623     _type
624     dotdotdot      uint8
625     in              []byte
626     out             []byte
627 }
628
629
630
631
632
633

```

```
634
635
636
637
638
639
640 type itab struct {
641     inter    *interfacetype
642     _type    *_type
643     link     *itab
644     bad      int
645     unused   int
646     fun      [0]func()
647 }
648
649 var hash     [1009]*itab
650 var ifacelock lock
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671 type hash_iter_sub struct {
672     e        *hash_entry
673     start    *hash_entry
674     last     *hash_entry
675 }
676
677 type hash_iter struct {
678     data     *uint8
679     elemsize int
680     changes  int
681     i        int
682     cycled   uint8
683     last_hash uint64
```

```
684         cycle    uint64
685         h         *hmap
686         t         *maptype
687         subtable_state [4]hash_iter_sub
688     }
689
690
691
692
693
694
695
696
697
698
699
700 type hmap struct {
701     count    uint
702     datasize    uint8
703     max_power    uint8
704     indirectval    uint8
705     valoff    uint8
706     changes    int
707     hash0    uint64
708     st         *hash_subtable
709 }
710
711 type hash_entry struct {
712     hash    uint64
713     data    [1]uint8
714 }
715
716 type hash_subtable struct {
717     power    uint8
718     used    uint8
719     datasize    uint8
720     max_probes    uint8
721     limit_bytes    int16
722     last    *hash_entry
723     entry    [1]hash_entry
724 }
725
726
727
728
729
730
731
732
```

```
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756 type sudog struct {
757     g          *g
758     selgen    uint
759     link      *sudog
760     elem      *uint8
761 }
762
763 type waitq struct {
764     first     *sudog
765     last      *sudog
766 }
767
768 type hchan struct {
769     qcount    uint
770     dataqsiz  uint
771     elemsize  uint16
772     closed    uint8
773     elemalign uint8
774     elemalg   *alg
775     sendx     uint
776     recvx     uint
777     recvq     waitq
778     sendq     waitq
779     lock
780 }
781
```

```
782 type scase struct {
783     sg      sudog
784     _chan   *hchan
785     pc      *uint8
786     kind    uint16
787     so      uint16
788     receivedp      *uint8
789 }
790
791 type _select struct {
792     tcase    uint16
793     ncase    uint16
794     pollorder      *uint16
795     lockorder      **hchan
796     scase    [1]scase
797 }
798
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/runtime/zversion.go

```
1 // auto generated by go tool dist
2
3 package runtime
4
5 const defaultGoroot = `/tmp/adg/godoc/go`
6 const theVersion = `go1.0.1`
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/runtime/cgo/cgo.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6 Package cgo contains runtime support for code generated
7 by the cgo tool. See the documentation for the cgo command
8 for details on using cgo.
9 */
10 package cgo
11
12 /*
13
14 #cgo darwin LDFLAGS: -lpthread
15 #cgo freebsd LDFLAGS: -lpthread
16 #cgo linux LDFLAGS: -lpthread
17 #cgo netbsd LDFLAGS: -lpthread
18 #cgo openbsd LDFLAGS: -lpthread
19 #cgo windows LDFLAGS: -lm -mthreads
20
21 */
22 import "C"
23
24 // Supports _cgo_panic by converting a string constant to an
25 // interface.
26
27 func cgoStringToEface(s string, ret *interface{}) {
28     *ret = s
29 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/runtime/debug/stack.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package debug contains facilities for programs to debug t
6 // they are running.
7 package debug
8
9 import (
10     "bytes"
11     "fmt"
12     "io/ioutil"
13     "os"
14     "runtime"
15 )
16
17 var (
18     dunno      = []byte("???" )
19     centerDot  = []byte(".")
20     dot        = []byte(".")
21 )
22
23 // PrintStack prints to standard error the stack trace retur
24 func PrintStack() {
25     os.Stderr.Write(stack())
26 }
27
28 // Stack returns a formatted stack trace of the goroutine th
29 // For each routine, it includes the source line information
30 // then attempts to discover, for Go functions, the calling
31 // method and the text of the line containing the invocation
32 func Stack() []byte {
33     return stack()
34 }
35
36 // stack implements Stack, skipping 2 frames
37 func stack() []byte {
38     buf := new(bytes.Buffer) // the returned data
39     // As we loop, we open files and read them. These va
40     // loaded file.
41     var lines [][]byte
```

```

42     var lastFile string
43     for i := 2; ; i++ { // Caller we care about is the u
44         pc, file, line, ok := runtime.Caller(i)
45         if !ok {
46             break
47         }
48         // Print this much at least. If we can't fi
49         fmt.Fprintf(buf, "%s:%d (0x%x)\n", file, lin
50         if file != lastFile {
51             data, err := ioutil.ReadFile(file)
52             if err != nil {
53                 continue
54             }
55             lines = bytes.Split(data, []byte{'\n
56             lastFile = file
57         }
58         line-- // in stack trace, lines are 1-indexe
59         fmt.Fprintf(buf, "\t%s: %s\n", function(pc),
60     }
61     return buf.Bytes()
62 }
63
64 // source returns a space-trimmed slice of the n'th line.
65 func source(lines [][]byte, n int) []byte {
66     if n < 0 || n >= len(lines) {
67         return dunno
68     }
69     return bytes.Trim(lines[n], " \t")
70 }
71
72 // function returns, if possible, the name of the function c
73 func function(pc uintptr) []byte {
74     fn := runtime.FuncForPC(pc)
75     if fn == nil {
76         return dunno
77     }
78     name := []byte(fn.Name())
79     // The name includes the path name to the package, w
80     // since the file name is already included. Plus, i
81     // That is, we see
82     //     runtime/debug.*T.ptrmethod
83     // and want
84     //     *T.ptrmethod
85     if period := bytes.Index(name, dot); period >= 0 {
86         name = name[period+1:]
87     }
88     name = bytes.Replace(name, centerDot, dot, -1)
89     return name
90 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/runtime/pprof/pprof.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package pprof writes runtime profiling data in the format
6 // by the pprof visualization tool.
7 // For more information about pprof, see
8 // http://code.google.com/p/google-perfctools/.
9 package pprof
10
11 import (
12     "bufio"
13     "bytes"
14     "fmt"
15     "io"
16     "runtime"
17     "sort"
18     "strings"
19     "sync"
20     "text/tabwriter"
21 )
22
23 // BUG(rsc): A bug in the OS X Snow Leopard 64-bit kernel pr
24 // CPU profiling from giving accurate results on that system
25
26 // A Profile is a collection of stack traces showing the cal
27 // that led to instances of a particular event, such as allo
28 // Packages can create and maintain their own profiles; the
29 // use is for tracking resources that must be explicitly clo
30 // or network connections.
31 //
32 // A Profile's methods can be called from multiple goroutine
33 //
34 // Each Profile has a unique name. A few profiles are prede
35 //
36 //     goroutine    - stack traces of all current goroutine
37 //     heap         - a sampling of all heap allocations
38 //     threadcreate - stack traces that led to the creation
39 //
40 // These predefine profiles maintain themselves and panic on
41 // Add or Remove method call.
```

```

42 //
43 // The CPU profile is not available as a Profile. It has a
44 // the StartCPUProfile and StopCPUProfile functions, because
45 // output to a writer during profiling.
46 //
47 type Profile struct {
48     name string
49     mu    sync.Mutex
50     m     map[interface{}][[]uintptr
51     count func() int
52     write func(io.Writer, int) error
53 }
54
55 // profiles records all registered profiles.
56 var profiles struct {
57     mu sync.Mutex
58     m  map[string]*Profile
59 }
60
61 var goroutineProfile = &Profile{
62     name: "goroutine",
63     count: countGoroutine,
64     write: writeGoroutine,
65 }
66
67 var threadcreateProfile = &Profile{
68     name: "threadcreate",
69     count: countThreadCreate,
70     write: writeThreadCreate,
71 }
72
73 var heapProfile = &Profile{
74     name: "heap",
75     count: countHeap,
76     write: writeHeap,
77 }
78
79 func lockProfiles() {
80     profiles.mu.Lock()
81     if profiles.m == nil {
82         // Initial built-in profiles.
83         profiles.m = map[string]*Profile{
84             "goroutine": goroutineProfile,
85             "threadcreate": threadcreateProfile,
86             "heap": heapProfile,
87         }
88     }
89 }
90
91 func unlockProfiles() {

```

```

92         profiles.mu.Unlock()
93     }
94
95     // NewProfile creates a new profile with the given name.
96     // If a profile with that name already exists, NewProfile pa
97     // The convention is to use a 'import/path.' prefix to creat
98     // separate name spaces for each package.
99     func NewProfile(name string) *Profile {
100         lockProfiles()
101         defer unlockProfiles()
102         if name == "" {
103             panic("pprof: NewProfile with empty name")
104         }
105         if profiles.m[name] != nil {
106             panic("pprof: NewProfile name already in use")
107         }
108         p := &Profile{
109             name: name,
110             m:     map[interface{}][]uintptr{},
111         }
112         profiles.m[name] = p
113         return p
114     }
115
116     // Lookup returns the profile with the given name, or nil if
117     func Lookup(name string) *Profile {
118         lockProfiles()
119         defer unlockProfiles()
120         return profiles.m[name]
121     }
122
123     // Profiles returns a slice of all the known profiles, sorte
124     func Profiles() []*Profile {
125         lockProfiles()
126         defer unlockProfiles()
127
128         var all []*Profile
129         for _, p := range profiles.m {
130             all = append(all, p)
131         }
132
133         sort.Sort(byName(all))
134         return all
135     }
136
137     type byName []*Profile
138
139     func (x byName) Len() int           { return len(x) }
140     func (x byName) Swap(i, j int)     { x[i], x[j] = x[j], x[i] }

```

```

141 func (x byName) Less(i, j int) bool { return x[i].name < x[j]
142
143 // Name returns this profile's name, which can be passed to
144 func (p *Profile) Name() string {
145     return p.name
146 }
147
148 // Count returns the number of execution stacks currently in
149 func (p *Profile) Count() int {
150     p.mu.Lock()
151     defer p.mu.Unlock()
152     if p.count != nil {
153         return p.count()
154     }
155     return len(p.m)
156 }
157
158 // Add adds the current execution stack to the profile, asso
159 // Add stores value in an internal map, so value must be sui
160 // a map key and will not be garbage collected until the cor
161 // call to Remove. Add panics if the profile already contai
162 //
163 // The skip parameter has the same meaning as runtime.Caller
164 // and controls where the stack trace begins. Passing skip=
165 // trace in the function calling Add. For example, given th
166 // execution stack:
167 //
168 //     Add
169 //     called from rpc.NewClient
170 //     called from mypkg.Run
171 //     called from main.main
172 //
173 // Passing skip=0 begins the stack trace at the call to Add
174 // Passing skip=1 begins the stack trace at the call to NewC
175 //
176 func (p *Profile) Add(value interface{}, skip int) {
177     if p.name == "" {
178         panic("pprof: use of uninitialized Profile")
179     }
180     if p.write != nil {
181         panic("pprof: Add called on built-in Profile")
182     }
183
184     stk := make([]uintptr, 32)
185     n := runtime.Callers(skip+1, stk[:])
186
187     p.mu.Lock()
188     defer p.mu.Unlock()
189     if p.m[value] != nil {

```

```

190             panic("pprof: Profile.Add of duplicate value
191         }
192         p.m[value] = stk[:n]
193     }
194
195     // Remove removes the execution stack associated with value
196     // It is a no-op if the value is not in the profile.
197     func (p *Profile) Remove(value interface{}) {
198         p.mu.Lock()
199         defer p.mu.Unlock()
200         delete(p.m, value)
201     }
202
203     // WriteTo writes a pprof-formatted snapshot of the profile
204     // If a write to w returns an error, WriteTo returns that er
205     // Otherwise, WriteTo returns nil.
206     //
207     // The debug parameter enables additional output.
208     // Passing debug=0 prints only the hexadecimal addresses tha
209     // Passing debug=1 adds comments translating addresses to fu
210     // and line numbers, so that a programmer can read the profi
211     //
212     // The predefined profiles may assign meaning to other debug
213     // for example, when printing the "goroutine" profile, debug
214     // print the goroutine stacks in the same form that a Go pro
215     // when dying due to an unrecovered panic.
216     func (p *Profile) WriteTo(w io.Writer, debug int) error {
217         if p.name == "" {
218             panic("pprof: use of zero Profile")
219         }
220         if p.write != nil {
221             return p.write(w, debug)
222         }
223
224         // Obtain consistent snapshot under lock; then proce
225         var all [][]uintptr
226         p.mu.Lock()
227         for _, stk := range p.m {
228             all = append(all, stk)
229         }
230         p.mu.Unlock()
231
232         // Map order is non-deterministic; make output deter
233         sort.Sort(stackProfile(all))
234
235         return printCountProfile(w, debug, p.name, stackProf
236     }
237
238     type stackProfile [][]uintptr
239

```

```

240 func (x stackProfile) Len() int           { return len(x)
241 func (x stackProfile) Stack(i int) []uintptr { return x[i] }
242 func (x stackProfile) Swap(i, j int)      { x[i], x[j] =
243 func (x stackProfile) Less(i, j int) bool {
244     t, u := x[i], x[j]
245     for k := 0; k < len(t) && k < len(u); k++ {
246         if t[k] != u[k] {
247             return t[k] < u[k]
248         }
249     }
250     return len(t) < len(u)
251 }
252
253 // A countProfile is a set of stack traces to be printed as
254 // grouped by stack trace. There are multiple implementatio
255 // all that matters is that we can find out how many traces
256 // and obtain each trace in turn.
257 type countProfile interface {
258     Len() int
259     Stack(i int) []uintptr
260 }
261
262 // printCountProfile prints a countProfile at the specified
263 func printCountProfile(w io.Writer, debug int, name string,
264     b := bufio.NewWriter(w)
265     var tw *tabwriter.Writer
266     w = b
267     if debug > 0 {
268         tw = tabwriter.NewWriter(w, 1, 8, 1, '\t', 0
269         w = tw
270     }
271
272     fmt.Fprintf(w, "%s profile: total %d\n", name, p.Len
273
274     // Build count of each stack.
275     var buf bytes.Buffer
276     key := func(stk []uintptr) string {
277         buf.Reset()
278         fmt.Fprintf(&buf, "@")
279         for _, pc := range stk {
280             fmt.Fprintf(&buf, " %#x", pc)
281         }
282         return buf.String()
283     }
284     m := map[string]int{}
285     n := p.Len()
286     for i := 0; i < n; i++ {
287         m[key(p.Stack(i))]++
288     }

```

```

289
290 // Print stacks, listing count on first occurrence o
291 for i := 0; i < n; i++ {
292     stk := p.Stack(i)
293     s := key(stk)
294     if count := m[s]; count != 0 {
295         fmt.Fprintf(w, "%d %s\n", count, s)
296         if debug > 0 {
297             printStackRecord(w, stk, fal
298         }
299         delete(m, s)
300     }
301 }
302
303 if tw != nil {
304     tw.Flush()
305 }
306 return b.Flush()
307 }
308
309 // printStackRecord prints the function + source line inform
310 // for a single stack trace.
311 func printStackRecord(w io.Writer, stk []uintptr, allFrames
312     show := allFrames
313     for _, pc := range stk {
314         f := runtime.FuncForPC(pc)
315         if f == nil {
316             show = true
317             fmt.Fprintf(w, "#\t%#x\n", pc)
318         } else {
319             file, line := f.FileLine(pc)
320             name := f.Name()
321             // Hide runtime.goexit and any runti
322             // This is useful mainly for allocat
323             if name == "runtime.goexit" || !show
324                 continue
325         }
326         show = true
327         fmt.Fprintf(w, "#\t%#x\t%s+ %#x\t%s:%
328     }
329 }
330 if !show {
331     // We didn't print anything; do it again,
332     // and this time include runtime functions.
333     printStackRecord(w, stk, true)
334     return
335 }
336 fmt.Fprintf(w, "\n")
337 }

```

```

338
339 // Interface to system profiles.
340
341 type byInUseBytes []runtime.MemProfileRecord
342
343 func (x byInUseBytes) Len() int           { return len(x) }
344 func (x byInUseBytes) Swap(i, j int)     { x[i], x[j] = x[j]
345 func (x byInUseBytes) Less(i, j int) bool { return x[i].InUs
346
347 // WriteHeapProfile is shorthand for Lookup("heap").WriteTo(
348 // It is preserved for backwards compatibility.
349 func WriteHeapProfile(w io.Writer) error {
350     return writeHeap(w, 0)
351 }
352
353 // countHeap returns the number of records in the heap profi
354 func countHeap() int {
355     n, _ := runtime.MemProfile(nil, false)
356     return n
357 }
358
359 // writeHeapProfile writes the current runtime heap profile
360 func writeHeap(w io.Writer, debug int) error {
361     // Find out how many records there are (MemProfile(n
362     // allocate that many records, and get the data.
363     // There's a race—more records might be added betwee
364     // the two calls—so allocate a few extra records for
365     // and also try again if we're very unlucky.
366     // The loop should only execute one iteration in the
367     var p []runtime.MemProfileRecord
368     n, ok := runtime.MemProfile(nil, false)
369     for {
370         // Allocate room for a slightly bigger profi
371         // in case a few more entries have been adde
372         // since the call to MemProfile.
373         p = make([]runtime.MemProfileRecord, n+50)
374         n, ok = runtime.MemProfile(p, false)
375         if ok {
376             p = p[0:n]
377             break
378         }
379         // Profile grew; try again.
380     }
381
382     sort.Sort(byInUseBytes(p))
383
384     b := bufio.NewWriter(w)
385     var tw *tabwriter.Writer
386     w = b
387     if debug > 0 {

```

```

388         tw = tabwriter.NewWriter(w, 1, 8, 1, '\t', 0
389         w = tw
390     }
391
392     var total runtime.MemProfileRecord
393     for i := range p {
394         r := &p[i]
395         total.AllocBytes += r.AllocBytes
396         total.AllocObjects += r.AllocObjects
397         total.FreeBytes += r.FreeBytes
398         total.FreeObjects += r.FreeObjects
399     }
400
401     // Technically the rate is MemProfileRate not 2*MemP
402     // but early versions of the C++ heap profiler repor
403     // so that's what pprof has come to expect.
404     fmt.Fprintf(w, "heap profile: %d: %d [%d: %d] @ heap
405         total.InUseObjects(), total.InUseBytes(),
406         total.AllocObjects, total.AllocBytes,
407         2*runtime.MemProfileRate)
408
409     for i := range p {
410         r := &p[i]
411         fmt.Fprintf(w, "%d: %d [%d: %d] @",
412             r.InUseObjects(), r.InUseBytes(),
413             r.AllocObjects, r.AllocBytes)
414         for _, pc := range r.Stack() {
415             fmt.Fprintf(w, " %#x", pc)
416         }
417         fmt.Fprintf(w, "\n")
418         if debug > 0 {
419             printStackRecord(w, r.Stack(), false
420         }
421     }
422
423     // Print memstats information too.
424     // Pprof will ignore, but useful for people
425     if debug > 0 {
426         s := new(runtime.MemStats)
427         runtime.ReadMemStats(s)
428         fmt.Fprintf(w, "\n# runtime.MemStats\n")
429         fmt.Fprintf(w, "# Alloc = %d\n", s.Alloc)
430         fmt.Fprintf(w, "# TotalAlloc = %d\n", s.Tota
431         fmt.Fprintf(w, "# Sys = %d\n", s.Sys)
432         fmt.Fprintf(w, "# Lookups = %d\n", s.Lookups
433         fmt.Fprintf(w, "# Mallocs = %d\n", s.Mallocs
434
435         fmt.Fprintf(w, "# HeapAlloc = %d\n", s.HeapA
436         fmt.Fprintf(w, "# HeapSys = %d\n", s.HeapSys

```

```

437         fmt.Fprintf(w, "# HeapIdle = %d\n", s.HeapId
438         fmt.Fprintf(w, "# HeapInuse = %d\n", s.HeapI
439
440         fmt.Fprintf(w, "# Stack = %d / %d\n", s.Stac
441         fmt.Fprintf(w, "# MSpan = %d / %d\n", s.MSpa
442         fmt.Fprintf(w, "# MCache = %d / %d\n", s.MCa
443         fmt.Fprintf(w, "# BuckHashSys = %d\n", s.Buc
444
445         fmt.Fprintf(w, "# NextGC = %d\n", s.NextGC)
446         fmt.Fprintf(w, "# PauseNs = %d\n", s.PauseNs
447         fmt.Fprintf(w, "# NumGC = %d\n", s.NumGC)
448         fmt.Fprintf(w, "# EnableGC = %v\n", s.Enable
449         fmt.Fprintf(w, "# DebugGC = %v\n", s.DebugGC
450     }
451
452     if tw != nil {
453         tw.Flush()
454     }
455     return b.Flush()
456 }
457
458 // countThreadCreate returns the size of the current ThreadC
459 func countThreadCreate() int {
460     n, _ := runtime.ThreadCreateProfile(nil)
461     return n
462 }
463
464 // writeThreadCreate writes the current runtime ThreadCreate
465 func writeThreadCreate(w io.Writer, debug int) error {
466     return writeRuntimeProfile(w, debug, "threadcreate",
467 }
468
469 // countGoroutine returns the number of goroutines.
470 func countGoroutine() int {
471     return runtime.NumGoroutine()
472 }
473
474 // writeGoroutine writes the current runtime GoroutineProfil
475 func writeGoroutine(w io.Writer, debug int) error {
476     if debug >= 2 {
477         return writeGoroutineStacks(w)
478     }
479     return writeRuntimeProfile(w, debug, "goroutine", ru
480 }
481
482 func writeGoroutineStacks(w io.Writer) error {
483     // We don't know how big the buffer needs to be to c
484     // all the goroutines. Start with 1 MB and try a fe
485     // Give up and use a truncated trace if 64 MB is not

```

```

486     buf := make([]byte, 1<<20)
487     for i := 0; ; i++ {
488         n := runtime.Stack(buf, true)
489         if n < len(buf) {
490             buf = buf[:n]
491             break
492         }
493         if len(buf) >= 64<<20 {
494             // Filled 64 MB - stop there.
495             break
496         }
497         buf = make([]byte, 2*len(buf))
498     }
499     _, err := w.Write(buf)
500     return err
501 }
502
503 func writeRuntimeProfile(w io.Writer, debug int, name string
504     // Find out how many records there are (fetch(nil)),
505     // allocate that many records, and get the data.
506     // There's a race—more records might be added between
507     // the two calls—so allocate a few extra records for
508     // and also try again if we're very unlucky.
509     // The loop should only execute one iteration in the
510     var p []runtime.StackRecord
511     n, ok := fetch(nil)
512     for {
513         // Allocate room for a slightly bigger profile
514         // in case a few more entries have been added
515         // since the call to ThreadProfile.
516         p = make([]runtime.StackRecord, n+10)
517         n, ok = fetch(p)
518         if ok {
519             p = p[0:n]
520             break
521         }
522         // Profile grew; try again.
523     }
524
525     return printCountProfile(w, debug, name, runtimeProfile)
526 }
527
528 type runtimeProfile []runtime.StackRecord
529
530 func (p runtimeProfile) Len() int           { return len(p) }
531 func (p runtimeProfile) Stack(i int) []uintptr { return p[i] }
532
533 var cpu struct {
534     sync.Mutex
535     profiling bool

```

```

536         done         chan bool
537     }
538
539     // StartCPUProfile enables CPU profiling for the current pro
540     // While profiling, the profile will be buffered and written
541     // StartCPUProfile returns an error if profiling is already
542     func StartCPUProfile(w io.Writer) error {
543         // The runtime routines allow a variable profiling r
544         // but in practice operating systems cannot trigger
545         // at more than about 500 Hz, and our processing of
546         // signal is not cheap (mostly getting the stack tra
547         // 100 Hz is a reasonable choice: it is frequent eno
548         // produce useful data, rare enough not to bog down
549         // system, and a nice round number to make it easy t
550         // convert sample counts to seconds. Instead of req
551         // each client to specify the frequency, we hard cod
552         const hz = 100
553
554         // Avoid queueing behind StopCPUProfile.
555         // Could use TryLock instead if we had it.
556         if cpu.profiling {
557             return fmt.Errorf("cpu profiling already in
558         }
559
560         cpu.Lock()
561         defer cpu.Unlock()
562         if cpu.done == nil {
563             cpu.done = make(chan bool)
564         }
565         // Double-check.
566         if cpu.profiling {
567             return fmt.Errorf("cpu profiling already in
568         }
569         cpu.profiling = true
570         runtime.SetCPUProfileRate(hz)
571         go profileWriter(w)
572         return nil
573     }
574
575     func profileWriter(w io.Writer) {
576         for {
577             data := runtime.CPUProfile()
578             if data == nil {
579                 break
580             }
581             w.Write(data)
582         }
583         cpu.done <- true
584     }

```

```
585
586 // StopCPUProfile stops the current CPU profile, if any.
587 // StopCPUProfile only returns after all the writes for the
588 // profile have completed.
589 func StopCPUProfile() {
590     cpu.Lock()
591     defer cpu.Unlock()
592
593     if !cpu.profiling {
594         return
595     }
596     cpu.profiling = false
597     runtime.SetCPUProfileRate(0)
598     <-cpu.done
599 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/sort/search.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // This file implements binary search.
6
7 package sort
8
9 // Search uses binary search to find and return the smallest
10 // in [0, n) at which f(i) is true, assuming that on the ran
11 // f(i) == true implies f(i+1) == true. That is, Search req
12 // f is false for some (possibly empty) prefix of the input
13 // and then true for the (possibly empty) remainder; Search
14 // the first true index. If there is no such index, Search
15 // Search calls f(i) only for i in the range [0, n).
16 //
17 // A common use of Search is to find the index i for a value
18 // a sorted, indexable data structure such as an array or sl
19 // In this case, the argument f, typically a closure, captur
20 // to be searched for, and how the data structure is indexed
21 // ordered.
22 //
23 // For instance, given a slice data sorted in ascending orde
24 // the call Search(len(data), func(i int) bool { return data
25 // returns the smallest index i such that data[i] >= 23. If
26 // wants to find whether 23 is in the slice, it must test da
27 // separately.
28 //
29 // Searching data sorted in descending order would use the <
30 // operator instead of the >= operator.
31 //
32 // To complete the example above, the following code tries t
33 // x in an integer slice data sorted in ascending order:
34 //
35 //     x := 23
36 //     i := sort.Search(len(data), func(i int) bool { retur
37 //     if i < len(data) && data[i] == x {
38 //         // x is present at data[i]
39 //     } else {
40 //         // x is not present in data,
41 //         // but i is the index where it would be inse
42 //     }
43 //
44 // As a more whimsical example, this program guesses your nu
```

```

45 //
46 //     func GuessingGame() {
47 //         var s string
48 //         fmt.Printf("Pick an integer from 0 to 100.\n
49 //         answer := sort.Search(100, func(i int) bool
50 //             fmt.Printf("Is your number <= %d? ",
51 //             fmt.Scanf("%s", &s)
52 //             return s != "" && s[0] == 'y'
53 //         })
54 //         fmt.Printf("Your number is %d.\n", answer)
55 //     }
56 //
57 func Search(n int, f func(int) bool) int {
58     // Define f(-1) == false and f(n) == true.
59     // Invariant: f(i-1) == false, f(j) == true.
60     i, j := 0, n
61     for i < j {
62         h := i + (j-i)/2 // avoid overflow when comp
63         // i ≤ h < j
64         if !f(h) {
65             i = h + 1 // preserves f(i-1) == fal
66         } else {
67             j = h // preserves f(j) == true
68         }
69     }
70     // i == j, f(i-1) == false, and f(j) (= f(i)) == tru
71     return i
72 }
73
74 // Convenience wrappers for common cases.
75
76 // SearchInts searches for x in a sorted slice of ints and r
77 // as specified by Search. The slice must be sorted in ascen
78 //
79 func SearchInts(a []int, x int) int {
80     return Search(len(a), func(i int) bool { return a[i]
81 }
82
83 // SearchFloat64s searches for x in a sorted slice of float6
84 // as specified by Search. The slice must be sorted in ascen
85 //
86 func SearchFloat64s(a []float64, x float64) int {
87     return Search(len(a), func(i int) bool { return a[i]
88 }
89
90 // SearchStrings searches for x slice a sorted slice of stri
91 // as specified by Search. The slice must be sorted in ascen
92 //
93 func SearchStrings(a []string, x string) int {
94     return Search(len(a), func(i int) bool { return a[i]

```

```
95 }
96
97 // Search returns the result of applying SearchInts to the r
98 func (p IntSlice) Search(x int) int { return SearchInts(p, x
99
100 // Search returns the result of applying SearchFloat64s to t
101 func (p Float64Slice) Search(x float64) int { return SearchF
102
103 // Search returns the result of applying SearchStrings to th
104 func (p StringSlice) Search(x string) int { return SearchStr
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/sort/sort.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package sort provides primitives for sorting slices and u
6 // collections.
7 package sort
8
9 import "math"
10
11 // A type, typically a collection, that satisfies sort.Interface
12 // sorted by the routines in this package. The methods requ
13 // elements of the collection be enumerated by an integer in
14 type Interface interface {
15     // Len is the number of elements in the collection.
16     Len() int
17     // Less returns whether the element with index i sho
18     // before the element with index j.
19     Less(i, j int) bool
20     // Swap swaps the elements with indexes i and j.
21     Swap(i, j int)
22 }
23
24 func min(a, b int) int {
25     if a < b {
26         return a
27     }
28     return b
29 }
30
31 // Insertion sort
32 func insertionSort(data Interface, a, b int) {
33     for i := a + 1; i < b; i++ {
34         for j := i; j > a && data.Less(j, j-1); j--
35             data.Swap(j, j-1)
36     }
37 }
38 }
39
40 // siftDown implements the heap property on data[lo, hi).
41 // first is an offset into the array where the root of the h
42 func siftDown(data Interface, lo, hi, first int) {
43     root := lo
44     for {
```

```

45         child := 2*root + 1
46         if child >= hi {
47             break
48         }
49         if child+1 < hi && data.Less(first+child, fi
50             child++
51         }
52         if !data.Less(first+root, first+child) {
53             return
54         }
55         data.Swap(first+root, first+child)
56         root = child
57     }
58 }
59
60 func heapSort(data Interface, a, b int) {
61     first := a
62     lo := 0
63     hi := b - a
64
65     // Build heap with greatest element at top.
66     for i := (hi - 1) / 2; i >= 0; i-- {
67         siftDown(data, i, hi, first)
68     }
69
70     // Pop elements, largest first, into end of data.
71     for i := hi - 1; i >= 0; i-- {
72         data.Swap(first, first+i)
73         siftDown(data, lo, i, first)
74     }
75 }
76
77 // Quicksort, following Bentley and McIlroy,
78 // ``Engineering a Sort Function,' ' SP&E November 1993.
79
80 // medianOfThree moves the median of the three values data[a
81 func medianOfThree(data Interface, a, b, c int) {
82     m0 := b
83     m1 := a
84     m2 := c
85     // bubble sort on 3 elements
86     if data.Less(m1, m0) {
87         data.Swap(m1, m0)
88     }
89     if data.Less(m2, m1) {
90         data.Swap(m2, m1)
91     }
92     if data.Less(m1, m0) {
93         data.Swap(m1, m0)
94     }

```

```

95         // now data[m0] <= data[m1] <= data[m2]
96     }
97
98     func swapRange(data Interface, a, b, n int) {
99         for i := 0; i < n; i++ {
100             data.Swap(a+i, b+i)
101         }
102     }
103
104     func doPivot(data Interface, lo, hi int) (midlo, midhi int)
105         m := lo + (hi-lo)/2 // Written like this to avoid in
106         if hi-lo > 40 {
107             // Tukey's ``Ninther,'' median of three medi
108             s := (hi - lo) / 8
109             medianOfThree(data, lo, lo+s, lo+2*s)
110             medianOfThree(data, m, m-s, m+s)
111             medianOfThree(data, hi-1, hi-1-s, hi-1-2*s)
112         }
113         medianOfThree(data, lo, m, hi-1)
114
115         // Invariants are:
116         //     data[lo] = pivot (set up by ChoosePivot)
117         //     data[lo <= i < a] = pivot
118         //     data[a <= i < b] < pivot
119         //     data[b <= i < c] is unexamined
120         //     data[c <= i < d] > pivot
121         //     data[d <= i < hi] = pivot
122         //
123         // Once b meets c, can swap the "= pivot" sections
124         // into the middle of the slice.
125         pivot := lo
126         a, b, c, d := lo+1, lo+1, hi, hi
127         for b < c {
128             if data.Less(b, pivot) { // data[b] < pivot
129                 b++
130                 continue
131             }
132             if !data.Less(pivot, b) { // data[b] = pivot
133                 data.Swap(a, b)
134                 a++
135                 b++
136                 continue
137             }
138             if data.Less(pivot, c-1) { // data[c-1] > pi
139                 c--
140                 continue
141             }
142             if !data.Less(c-1, pivot) { // data[c-1] = p
143                 data.Swap(c-1, d-1)

```

```

144             c--
145             d--
146             continue
147         }
148         // data[b] > pivot; data[c-1] < pivot
149         data.Swap(b, c-1)
150         b++
151         c--
152     }
153
154     n := min(b-a, a-lo)
155     swapRange(data, lo, b-n, n)
156
157     n = min(hi-d, d-c)
158     swapRange(data, c, hi-n, n)
159
160     return lo + b - a, hi - (d - c)
161 }
162
163 func quickSort(data Interface, a, b, maxDepth int) {
164     for b-a > 7 {
165         if maxDepth == 0 {
166             heapSort(data, a, b)
167             return
168         }
169         maxDepth--
170         mlo, mhi := doPivot(data, a, b)
171         // Avoiding recursion on the larger subprobl
172         // a stack depth of at most lg(b-a).
173         if mlo-a < b-mhi {
174             quickSort(data, a, mlo, maxDepth)
175             a = mhi // i.e., quickSort(data, mhi
176         } else {
177             quickSort(data, mhi, b, maxDepth)
178             b = mlo // i.e., quickSort(data, a,
179         }
180     }
181     if b-a > 1 {
182         insertionSort(data, a, b)
183     }
184 }
185
186 // Sort sorts data.
187 // It makes one call to data.Len to determine n, and O(n*log
188 // data.Len and data.Swap. The sort is not guaranteed to be
189 func Sort(data Interface) {
190     // Switch to heapsort if depth of 2*ceil(lg(n+1)) is
191     n := data.Len()
192     maxDepth := 0

```

```

193         for i := n; i > 0; i >>= 1 {
194             maxDepth++
195         }
196         maxDepth *= 2
197         quickSort(data, 0, n, maxDepth)
198     }
199
200     // IsSorted reports whether data is sorted.
201     func IsSorted(data Interface) bool {
202         n := data.Len()
203         for i := n - 1; i > 0; i-- {
204             if data.Less(i, i-1) {
205                 return false
206             }
207         }
208         return true
209     }
210
211     // Convenience types for common cases
212
213     // IntSlice attaches the methods of Interface to []int, sort
214     type IntSlice []int
215
216     func (p IntSlice) Len() int           { return len(p) }
217     func (p IntSlice) Less(i, j int) bool { return p[i] < p[j] }
218     func (p IntSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
219
220     // Sort is a convenience method.
221     func (p IntSlice) Sort() { Sort(p) }
222
223     // Float64Slice attaches the methods of Interface to []float
224     type Float64Slice []float64
225
226     func (p Float64Slice) Len() int           { return len(p) }
227     func (p Float64Slice) Less(i, j int) bool { return p[i] < p[j] }
228     func (p Float64Slice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
229
230     // Sort is a convenience method.
231     func (p Float64Slice) Sort() { Sort(p) }
232
233     // StringSlice attaches the methods of Interface to []string
234     type StringSlice []string
235
236     func (p StringSlice) Len() int           { return len(p) }
237     func (p StringSlice) Less(i, j int) bool { return p[i] < p[j] }
238     func (p StringSlice) Swap(i, j int)      { p[i], p[j] = p[j], p[i] }
239
240     // Sort is a convenience method.
241     func (p StringSlice) Sort() { Sort(p) }
242

```

```
243 // Convenience wrappers for common cases
244
245 // Ints sorts a slice of ints in increasing order.
246 func Ints(a []int) { Sort(IntSlice(a)) }
247
248 // Float64s sorts a slice of float64s in increasing order.
249 func Float64s(a []float64) { Sort(Float64Slice(a)) }
250
251 // Strings sorts a slice of strings in increasing order.
252 func Strings(a []string) { Sort(StringSlice(a)) }
253
254 // IntsAreSorted tests whether a slice of ints is sorted in
255 func IntsAreSorted(a []int) bool { return IsSorted(IntSlice(
256
257 // Float64sAreSorted tests whether a slice of float64s is so
258 func Float64sAreSorted(a []float64) bool { return IsSorted(F
259
260 // StringsAreSorted tests whether a slice of strings is sort
261 func StringsAreSorted(a []string) bool { return IsSorted(Str
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strconv/atob.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package strconv
6
7 // ParseBool returns the boolean value represented by the st
8 // It accepts 1, t, T, TRUE, true, True, 0, f, F, FALSE, fal
9 // Any other value returns an error.
10 func ParseBool(str string) (value bool, err error) {
11     switch str {
12     case "1", "t", "T", "true", "TRUE", "True":
13         return true, nil
14     case "0", "f", "F", "false", "FALSE", "False":
15         return false, nil
16     }
17     return false, syntaxError("ParseBool", str)
18 }
19
20 // FormatBool returns "true" or "false" according to the val
21 func FormatBool(b bool) string {
22     if b {
23         return "true"
24     }
25     return "false"
26 }
27
28 // AppendBool appends "true" or "false", according to the va
29 // to dst and returns the extended buffer.
30 func AppendBool(dst []byte, b bool) []byte {
31     if b {
32         return append(dst, "true"... )
33     }
34     return append(dst, "false"... )
35 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strconv/atof.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package strconv implements conversions to and from string
6 // of basic data types.
7 package strconv
8
9 // decimal to binary floating point conversion.
10 // Algorithm:
11 //   1) Store input in multiprecision decimal.
12 //   2) Multiply/divide decimal by powers of two until in ra
13 //   3) Multiply by 2^precision and round to get mantissa.
14
15 import "math"
16
17 var optimize = true // can change for testing
18
19 func equalIgnoreCase(s1, s2 string) bool {
20     if len(s1) != len(s2) {
21         return false
22     }
23     for i := 0; i < len(s1); i++ {
24         c1 := s1[i]
25         if 'A' <= c1 && c1 <= 'Z' {
26             c1 += 'a' - 'A'
27         }
28         c2 := s2[i]
29         if 'A' <= c2 && c2 <= 'Z' {
30             c2 += 'a' - 'A'
31         }
32         if c1 != c2 {
33             return false
34         }
35     }
36     return true
37 }
38
39 func special(s string) (f float64, ok bool) {
40     switch {
41     case equalIgnoreCase(s, "nan"):
42         return math.NaN(), true
43     case equalIgnoreCase(s, "-inf"),
44         equalIgnoreCase(s, "-infinity"):
```

```

45         return math.Inf(-1), true
46     case equalIgnoreCase(s, "+inf"),
47         equalIgnoreCase(s, "+infinity"),
48         equalIgnoreCase(s, "inf"),
49         equalIgnoreCase(s, "infinity"):
50         return math.Inf(1), true
51     }
52     return
53 }
54
55 func (b *decimal) set(s string) (ok bool) {
56     i := 0
57     b.neg = false
58     b.trunc = false
59
60     // optional sign
61     if i >= len(s) {
62         return
63     }
64     switch {
65     case s[i] == '+':
66         i++
67     case s[i] == '-':
68         b.neg = true
69         i++
70     }
71
72     // digits
73     sawdot := false
74     sawdigits := false
75     for ; i < len(s); i++ {
76         switch {
77         case s[i] == '.':
78             if sawdot {
79                 return
80             }
81             sawdot = true
82             b.dp = b.nd
83             continue
84
85         case '0' <= s[i] && s[i] <= '9':
86             sawdigits = true
87             if s[i] == '0' && b.nd == 0 { // ign
88                 b.dp--
89                 continue
90             }
91             if b.nd < len(b.d) {
92                 b.d[b.nd] = s[i]
93                 b.nd++
94             } else if s[i] != '0' {

```

```

95             b.trunc = true
96         }
97         continue
98     }
99     break
100 }
101 if !sawdigits {
102     return
103 }
104 if !sawdot {
105     b.dp = b.nd
106 }
107
108 // optional exponent moves decimal point.
109 // if we read a very large, very long number,
110 // just be sure to move the decimal point by
111 // a lot (say, 100000). it doesn't matter if it's
112 // not the exact number.
113 if i < len(s) && (s[i] == 'e' || s[i] == 'E') {
114     i++
115     if i >= len(s) {
116         return
117     }
118     esign := 1
119     if s[i] == '+' {
120         i++
121     } else if s[i] == '-' {
122         i++
123         esign = -1
124     }
125     if i >= len(s) || s[i] < '0' || s[i] > '9' {
126         return
127     }
128     e := 0
129     for ; i < len(s) && '0' <= s[i] && s[i] <= '9' {
130         if e < 10000 {
131             e = e*10 + int(s[i]) - '0'
132         }
133     }
134     b.dp += e * esign
135 }
136
137 if i != len(s) {
138     return
139 }
140
141 ok = true
142 return
143 }

```

```

144
145 // decimal power of ten to binary power of two.
146 var powtab = []int{1, 3, 6, 9, 13, 16, 19, 23, 26}
147
148 func (d *decimal) floatBits(flt *floatInfo) (b uint64, overf
149     var exp int
150     var mant uint64
151
152     // Zero is always a special case.
153     if d.nd == 0 {
154         mant = 0
155         exp = flt.bias
156         goto out
157     }
158
159     // Obvious overflow/underflow.
160     // These bounds are for 64-bit floats.
161     // Will have to change if we want to support 80-bit
162     if d.dp > 310 {
163         goto overflow
164     }
165     if d.dp < -330 {
166         // zero
167         mant = 0
168         exp = flt.bias
169         goto out
170     }
171
172     // Scale by powers of two until in range [0.5, 1.0)
173     exp = 0
174     for d.dp > 0 {
175         var n int
176         if d.dp >= len(powtab) {
177             n = 27
178         } else {
179             n = powtab[d.dp]
180         }
181         d.Shift(-n)
182         exp += n
183     }
184     for d.dp < 0 || d.dp == 0 && d.d[0] < '5' {
185         var n int
186         if -d.dp >= len(powtab) {
187             n = 27
188         } else {
189             n = powtab[-d.dp]
190         }
191         d.Shift(n)
192         exp -= n

```

```

193     }
194
195     // Our range is [0.5,1) but floating point range is
196     exp--
197
198     // Minimum representable exponent is flt.bias+1.
199     // If the exponent is smaller, move it up and
200     // adjust d accordingly.
201     if exp < flt.bias+1 {
202         n := flt.bias + 1 - exp
203         d.Shift(-n)
204         exp += n
205     }
206
207     if exp-flt.bias >= 1<<flt.expbits-1 {
208         goto overflow
209     }
210
211     // Extract 1+flt.mantbits bits.
212     d.Shift(int(1 + flt.mantbits))
213     mant = d.RoundedInteger()
214
215     // Rounding might have added a bit; shift down.
216     if mant == 2<<flt.mantbits {
217         mant >>= 1
218         exp++
219         if exp-flt.bias >= 1<<flt.expbits-1 {
220             goto overflow
221         }
222     }
223
224     // Denormalized?
225     if mant&(1<<flt.mantbits) == 0 {
226         exp = flt.bias
227     }
228     goto out
229
230 overflow:
231     // ±Inf
232     mant = 0
233     exp = 1<<flt.expbits - 1 + flt.bias
234     overflow = true
235
236 out:
237     // Assemble bits.
238     bits := mant & (uint64(1)<<flt.mantbits - 1)
239     bits |= uint64((exp-flt.bias)&(1<<flt.expbits-1)) <<
240     if d.neg {
241         bits |= 1 << flt.mantbits << flt.expbits
242     }

```

```

243         return bits, overflow
244     }
245
246 // Compute exact floating-point integer from d's digits.
247 // Caller is responsible for avoiding overflow.
248 func (d *decimal) atof64int() float64 {
249     f := 0.0
250     for i := 0; i < d.nd; i++ {
251         f = f*10 + float64(d.d[i]-'0')
252     }
253     if d.neg {
254         f = -f
255     }
256     return f
257 }
258
259 func (d *decimal) atof32int() float32 {
260     f := float32(0)
261     for i := 0; i < d.nd; i++ {
262         f = f*10 + float32(d.d[i]-'0')
263     }
264     if d.neg {
265         f = -f
266     }
267     return f
268 }
269
270 // Reads a uint64 decimal mantissa, which might be truncated
271 func (d *decimal) atou64() (mant uint64, digits int) {
272     const uint64digits = 19
273     for i, c := range d.d[:d.nd] {
274         if i == uint64digits {
275             return mant, i
276         }
277         mant = 10*mant + uint64(c-'0')
278     }
279     return mant, d.nd
280 }
281
282 // Exact powers of 10.
283 var float64pow10 = []float64{
284     1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9,
285     1e10, 1e11, 1e12, 1e13, 1e14, 1e15, 1e16, 1e17, 1e18
286     1e20, 1e21, 1e22,
287 }
288 var float32pow10 = []float32{1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1
289
290 // If possible to convert decimal d to 64-bit float f exactl
291 // entirely in floating-point math, do so, avoiding the expe

```

```

292 // Three common cases:
293 //     value is exact integer
294 //     value is exact integer * exact power of ten
295 //     value is exact integer / exact power of ten
296 // These all produce potentially inexact but correctly round
297 func (d *decimal) atof64() (f float64, ok bool) {
298     // Exact integers are <= 10^15.
299     // Exact powers of ten are <= 10^22.
300     if d.nd > 15 {
301         return
302     }
303     switch {
304     case d.dp == d.nd: // int
305         f := d.atof64int()
306         return f, true
307
308     case d.dp > d.nd && d.dp <= 15+22: // int * 10^k
309         f := d.atof64int()
310         k := d.dp - d.nd
311         // If exponent is big but number of digits i
312         // can move a few zeros into the integer part
313         if k > 22 {
314             f *= float64pow10[k-22]
315             k = 22
316         }
317         return f * float64pow10[k], true
318
319     case d.dp < d.nd && d.nd-d.dp <= 22: // int / 10^k
320         f := d.atof64int()
321         return f / float64pow10[d.nd-d.dp], true
322     }
323     return
324 }
325
326 // If possible to convert decimal d to 32-bit float f exactl
327 // entirely in floating-point math, do so, avoiding the mach
328 func (d *decimal) atof32() (f float32, ok bool) {
329     // Exact integers are <= 10^7.
330     // Exact powers of ten are <= 10^10.
331     if d.nd > 7 {
332         return
333     }
334     switch {
335     case d.dp == d.nd: // int
336         f := d.atof32int()
337         return f, true
338
339     case d.dp > d.nd && d.dp <= 7+10: // int * 10^k
340         f := d.atof32int()

```

```

341         k := d.dp - d.nd
342         // If exponent is big but number of digits i
343         // can move a few zeros into the integer part
344         if k > 10 {
345             f *= float32pow10[k-10]
346             k = 10
347         }
348         return f * float32pow10[k], true
349
350     case d.dp < d.nd && d.nd-d.dp <= 10: // int / 10^k
351         f := d.atof32int()
352         return f / float32pow10[d.nd-d.dp], true
353     }
354     return
355 }
356
357 const fnParseFloat = "ParseFloat"
358
359 func atof32(s string) (f float32, err error) {
360     if val, ok := special(s); ok {
361         return float32(val), nil
362     }
363
364     var d decimal
365     if !d.set(s) {
366         return 0, syntaxError(fnParseFloat, s)
367     }
368     if optimize {
369         if f, ok := d.atof32(); ok {
370             return f, nil
371         }
372     }
373     b, ovf := d.floatBits(&float32info)
374     f = math.Float32frombits(uint32(b))
375     if ovf {
376         err = rangeError(fnParseFloat, s)
377     }
378     return f, err
379 }
380
381 func atof64(s string) (f float64, err error) {
382     if val, ok := special(s); ok {
383         return val, nil
384     }
385
386     var d decimal
387     if !d.set(s) {
388         return 0, syntaxError(fnParseFloat, s)
389     }
390     if optimize {

```

```

391         if f, ok := d atof64(); ok {
392             return f, nil
393         }
394
395         // Try another fast path.
396         ext := new(extFloat)
397         if ok := ext.AssignDecimal(&d); ok {
398             b, ovf := ext.floatBits()
399             f = math.Float64frombits(b)
400             if ovf {
401                 err = rangeError(fnParseFtoa
402             }
403             return f, err
404         }
405     }
406     b, ovf := d.floatBits(&float64info)
407     f = math.Float64frombits(b)
408     if ovf {
409         err = rangeError(fnParseFloat, s)
410     }
411     return f, err
412 }
413
414 // ParseFloat converts the string s to a floating-point numb
415 // with the precision specified by bitSize: 32 for float32,
416 // When bitSize=32, the result still has type float64, but i
417 // convertible to float32 without changing its value.
418 //
419 // If s is well-formed and near a valid floating point numbe
420 // ParseFloat returns the nearest floating point number roun
421 // using IEEE754 unbiased rounding.
422 //
423 // The errors that ParseFloat returns have concrete type *Nu
424 // and include err.Num = s.
425 //
426 // If s is not syntactically well-formed, ParseFloat returns
427 //
428 // If s is syntactically well-formed but is more than 1/2 UL
429 // away from the largest floating point number of the given
430 // ParseFloat returns f = ±Inf, err.Error = ErrRange.
431 func ParseFloat(s string, bitSize int) (f float64, err error
432     if bitSize == 32 {
433         f1, err1 := atof32(s)
434         return float64(f1), err1
435     }
436     f1, err1 := atof64(s)
437     return f1, err1
438 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strconv/atoi.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package strconv
6
7 import "errors"
8
9 // ErrRange indicates that a value is out of range for the t
10 var ErrRange = errors.New("value out of range")
11
12 // ErrSyntax indicates that a value does not have the right
13 var ErrSyntax = errors.New("invalid syntax")
14
15 // A NumError records a failed conversion.
16 type NumError struct {
17     Func string // the failing function (ParseBool, Pars
18     Num  string // the input
19     Err  error  // the reason the conversion failed (Err
20 }
21
22 func (e *NumError) Error() string {
23     return "strconv." + e.Func + ": " + `parsing "` + e.
24 }
25
26 func syntaxError(fn, str string) *NumError {
27     return &NumError{fn, str, ErrSyntax}
28 }
29
30 func rangeError(fn, str string) *NumError {
31     return &NumError{fn, str, ErrRange}
32 }
33
34 const intSize = 32 << uint(^uint(0)>>63)
35
36 const IntSize = intSize // number of bits in int, uint (32 o
37
38 // Return the first number n such that n*base >= 1<<64.
39 func cutoff64(base int) uint64 {
40     if base < 2 {
41         return 0
42     }
43     return (1<<64-1)/uint64(base) + 1
44 }
```

```

45
46 // ParseUint is like ParseInt but for unsigned numbers.
47 func ParseUint(s string, b int, bitSize int) (n uint64, err
48     var cutoff, maxVal uint64
49
50     if bitSize == 0 {
51         bitSize = int(IntSize)
52     }
53
54     s0 := s
55     switch {
56     case len(s) < 1:
57         err = ErrSyntax
58         goto Error
59
60     case 2 <= b && b <= 36:
61         // valid base; nothing to do
62
63     case b == 0:
64         // Look for octal, hex prefix.
65         switch {
66         case s[0] == '0' && len(s) > 1 && (s[1] == '
67             b = 16
68             s = s[2:]
69             if len(s) < 1 {
70                 err = ErrSyntax
71                 goto Error
72             }
73         case s[0] == '0':
74             b = 8
75         default:
76             b = 10
77         }
78
79     default:
80         err = errors.New("invalid base " + Itoa(b))
81         goto Error
82     }
83
84     n = 0
85     cutoff = cutoff64(b)
86     maxVal = 1<<uint(bitSize) - 1
87
88     for i := 0; i < len(s); i++ {
89         var v byte
90         d := s[i]
91         switch {
92         case '0' <= d && d <= '9':
93             v = d - '0'
94         case 'a' <= d && d <= 'z':

```

```

95         v = d - 'a' + 10
96     case 'A' <= d && d <= 'Z':
97         v = d - 'A' + 10
98     default:
99         n = 0
100        err = ErrSyntax
101        goto Error
102    }
103    if int(v) >= b {
104        n = 0
105        err = ErrSyntax
106        goto Error
107    }
108
109    if n >= cutoff {
110        // n*b overflows
111        n = 1<<64 - 1
112        err = ErrRange
113        goto Error
114    }
115    n *= uint64(b)
116
117    n1 := n + uint64(v)
118    if n1 < n || n1 > maxVal {
119        // n+v overflows
120        n = 1<<64 - 1
121        err = ErrRange
122        goto Error
123    }
124    n = n1
125 }
126
127     return n, nil
128
129 Error:
130     return n, &NumError{"ParseUint", s0, err}
131 }
132
133 // ParseInt interprets a string s in the given base (2 to 36
134 // returns the corresponding value i. If base == 0, the bas
135 // implied by the string's prefix: base 16 for "0x", base 8
136 // "0", and base 10 otherwise.
137 //
138 // The bitSize argument specifies the integer type
139 // that the result must fit into. Bit sizes 0, 8, 16, 32, a
140 // correspond to int, int8, int16, int32, and int64.
141 //
142 // The errors that ParseInt returns have concrete type *NumE
143 // and include err.Num = s. If s is empty or contains inval

```

```

144 // digits, err.Error = ErrSyntax; if the value corresponding
145 // to s cannot be represented by a signed integer of the
146 // given size, err.Error = ErrRange.
147 func ParseInt(s string, base int, bitSize int) (i int64, err
148     const fnParseInt = "ParseInt"
149
150     if bitSize == 0 {
151         bitSize = int(IntSize)
152     }
153
154     // Empty string bad.
155     if len(s) == 0 {
156         return 0, syntaxError(fnParseInt, s)
157     }
158
159     // Pick off leading sign.
160     s0 := s
161     neg := false
162     if s[0] == '+' {
163         s = s[1:]
164     } else if s[0] == '-' {
165         neg = true
166         s = s[1:]
167     }
168
169     // Convert unsigned and check range.
170     var un uint64
171     un, err = ParseUint(s, base, bitSize)
172     if err != nil && err.(*NumError).Err != ErrRange {
173         err.(*NumError).Func = fnParseInt
174         err.(*NumError).Num = s0
175         return 0, err
176     }
177     cutoff := uint64(1 << uint(bitSize-1))
178     if !neg && un >= cutoff {
179         return int64(cutoff - 1), rangeError(fnParse
180     }
181     if neg && un > cutoff {
182         return -int64(cutoff), rangeError(fnParseInt
183     }
184     n := int64(un)
185     if neg {
186         n = -n
187     }
188     return n, nil
189 }
190
191 // Atoi is shorthand for ParseInt(s, 10, 0).
192 func Atoi(s string) (i int, err error) {

```

```
193         i64, err := ParseInt(s, 10, 0)
194         return int(i64), err
195     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strconv/decimal.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Multiprecision decimal numbers.
6 // For floating-point formatting only; not general purpose.
7 // Only operations are assign and (binary) left/right shift.
8 // Can do binary floating point in multiprecision decimal pr
9 // because 2 divides 10; cannot do decimal floating point
10 // in multiprecision binary precisely.
11
12 package strconv
13
14 type decimal struct {
15     d      [800]byte // digits
16     nd     int      // number of digits used
17     dp     int      // decimal point
18     neg    bool
19     trunc bool // discarded nonzero digits beyond d[:nd]
20 }
21
22 func (a *decimal) String() string {
23     n := 10 + a.nd
24     if a.dp > 0 {
25         n += a.dp
26     }
27     if a.dp < 0 {
28         n += -a.dp
29     }
30
31     buf := make([]byte, n)
32     w := 0
33     switch {
34     case a.nd == 0:
35         return "0"
36
37     case a.dp <= 0:
38         // zeros fill space between decimal point an
39         buf[w] = '0'
40         w++
41         buf[w] = '.'
42         w++
43         w += digitZero(buf[w : w+-a.dp])
44         w += copy(buf[w:], a.d[0:a.nd])
```

```

45
46     case a.dp < a.nd:
47         // decimal point in middle of digits
48         w += copy(buf[w:], a.d[0:a.dp])
49         buf[w] = '.'
50         w++
51         w += copy(buf[w:], a.d[a.dp:a.nd])
52
53     default:
54         // zeros fill space between digits and decim
55         w += copy(buf[w:], a.d[0:a.nd])
56         w += digitZero(buf[w : w+a.dp-a.nd])
57     }
58     return string(buf[0:w])
59 }
60
61 func digitZero(dst []byte) int {
62     for i := range dst {
63         dst[i] = '0'
64     }
65     return len(dst)
66 }
67
68 // trim trailing zeros from number.
69 // (They are meaningless; the decimal point is tracked
70 // independent of the number of digits.)
71 func trim(a *decimal) {
72     for a.nd > 0 && a.d[a.nd-1] == '0' {
73         a.nd--
74     }
75     if a.nd == 0 {
76         a.dp = 0
77     }
78 }
79
80 // Assign v to a.
81 func (a *decimal) Assign(v uint64) {
82     var buf [50]byte
83
84     // Write reversed decimal in buf.
85     n := 0
86     for v > 0 {
87         v1 := v / 10
88         v -= 10 * v1
89         buf[n] = byte(v + '0')
90         n++
91         v = v1
92     }
93
94     // Reverse again to produce forward decimal in a.d.

```

```

95         a.nd = 0
96         for n--; n >= 0; n-- {
97             a.d[a.nd] = buf[n]
98             a.nd++
99         }
100        a.dp = a.nd
101        trim(a)
102    }
103
104    // Maximum shift that we can do in one pass without overflow
105    // Signed int has 31 bits, and we have to be able to accommo
106    const maxShift = 27
107
108    // Binary shift right (* 2) by k bits.  k <= maxShift to avo
109    func rightShift(a *decimal, k uint) {
110        r := 0 // read pointer
111        w := 0 // write pointer
112
113        // Pick up enough leading digits to cover first shif
114        n := 0
115        for ; n>>k == 0; r++ {
116            if r >= a.nd {
117                if n == 0 {
118                    // a == 0; shouldn't get her
119                    a.nd = 0
120                    return
121                }
122                for n>>k == 0 {
123                    n = n * 10
124                    r++
125                }
126                break
127            }
128            c := int(a.d[r])
129            n = n*10 + c - '0'
130        }
131        a.dp -= r - 1
132
133        // Pick up a digit, put down a digit.
134        for ; r < a.nd; r++ {
135            c := int(a.d[r])
136            dig := n >> k
137            n -= dig << k
138            a.d[w] = byte(dig + '0')
139            w++
140            n = n*10 + c - '0'
141        }
142
143        // Put down extra digits.

```

```

144     for n > 0 {
145         dig := n >> k
146         n -= dig << k
147         if w < len(a.d) {
148             a.d[w] = byte(dig + '0')
149             w++
150         } else if dig > 0 {
151             a.trunc = true
152         }
153         n = n * 10
154     }
155
156     a.nd = w
157     trim(a)
158 }
159
160 // Cheat sheet for left shift: table indexed by shift count
161 // number of new digits that will be introduced by that shift
162 //
163 // For example, leftcheats[4] = {2, "625"}. That means that
164 // if we are shifting by 4 (multiplying by 16), it will add
165 // when the string prefix is "625" through "999", and one fe
166 // if the string prefix is "000" through "624".
167 //
168 // Credit for this trick goes to Ken.
169
170 type leftCheat struct {
171     delta int // number of new digits
172     cutoff string // minus one digit if original < a.
173 }
174
175 var leftcheats = []leftCheat{
176     // Leading digits of  $1/2^i = 5^i$ .
177     //  $5^{23}$  is not an exact 64-bit floating point number
178     // so have to use bc for the math.
179     /*
180         seq 27 | sed 's/^/5^/' | bc |
181         awk 'BEGIN{ print "\tleftCheat{ 0, \"\" },"
182         {
183             log2 = log(2)/log(10)
184             printf("\tleftCheat{ %d, \"%s\" },\t
185                 int(log2*NR+1), $0, 2**NR)
186         }'
187     */
188     {0, ""},
189     {1, "5"}, // * 2
190     {1, "25"}, // * 4
191     {1, "125"}, // * 8
192     {2, "625"}, // * 16

```

```

193         {2, "3125"}, // * 32
194         {2, "15625"}, // * 64
195         {3, "78125"}, // * 128
196         {3, "390625"}, // * 256
197         {3, "1953125"}, // * 512
198         {4, "9765625"}, // * 1024
199         {4, "48828125"}, // * 2048
200         {4, "244140625"}, // * 4096
201         {4, "1220703125"}, // * 8192
202         {5, "6103515625"}, // * 16384
203         {5, "30517578125"}, // * 32768
204         {5, "152587890625"}, // * 65536
205         {6, "762939453125"}, // * 131072
206         {6, "3814697265625"}, // * 262144
207         {6, "19073486328125"}, // * 524288
208         {7, "95367431640625"}, // * 1048576
209         {7, "476837158203125"}, // * 2097152
210         {7, "2384185791015625"}, // * 4194304
211         {7, "11920928955078125"}, // * 8388608
212         {8, "59604644775390625"}, // * 16777216
213         {8, "298023223876953125"}, // * 33554432
214         {8, "1490116119384765625"}, // * 67108864
215         {9, "7450580596923828125"}, // * 134217728
216     }
217
218     // Is the leading prefix of b lexicographically less than s?
219     func prefixIsLessThan(b []byte, s string) bool {
220         for i := 0; i < len(s); i++ {
221             if i >= len(b) {
222                 return true
223             }
224             if b[i] != s[i] {
225                 return b[i] < s[i]
226             }
227         }
228         return false
229     }
230
231     // Binary shift left (/ 2) by k bits. k <= maxShift to avoi
232     func leftShift(a *decimal, k uint) {
233         delta := leftcheats[k].delta
234         if prefixIsLessThan(a.d[0:a.nd], leftcheats[k].cutof
235             delta--
236     }
237
238     r := a.nd // read index
239     w := a.nd + delta // write index
240     n := 0
241
242     // Pick up a digit, put down a digit.

```

```

243     for r--; r >= 0; r-- {
244         n += (int(a.d[r]) - '0') << k
245         quo := n / 10
246         rem := n - 10*quo
247         w--
248         if w < len(a.d) {
249             a.d[w] = byte(rem + '0')
250         } else if rem != 0 {
251             a.trunc = true
252         }
253         n = quo
254     }
255
256     // Put down extra digits.
257     for n > 0 {
258         quo := n / 10
259         rem := n - 10*quo
260         w--
261         if w < len(a.d) {
262             a.d[w] = byte(rem + '0')
263         } else if rem != 0 {
264             a.trunc = true
265         }
266         n = quo
267     }
268
269     a.nd += delta
270     if a.nd >= len(a.d) {
271         a.nd = len(a.d)
272     }
273     a.dp += delta
274     trim(a)
275 }
276
277 // Binary shift left (k > 0) or right (k < 0).
278 func (a *decimal) Shift(k int) {
279     switch {
280     case a.nd == 0:
281         // nothing to do: a == 0
282     case k > 0:
283         for k > maxShift {
284             leftShift(a, maxShift)
285             k -= maxShift
286         }
287         leftShift(a, uint(k))
288     case k < 0:
289         for k < -maxShift {
290             rightShift(a, maxShift)
291             k += maxShift

```

```

292         }
293         rightShift(a, uint(-k))
294     }
295 }
296
297 // If we chop a at nd digits, should we round up?
298 func shouldRoundUp(a *decimal, nd int) bool {
299     if nd < 0 || nd >= a.nd {
300         return false
301     }
302     if a.d[nd] == '5' && nd+1 == a.nd { // exactly halfw
303         // if we truncated, a little higher than wha
304         if a.trunc {
305             return true
306         }
307         return nd > 0 && (a.d[nd-1]-'0')%2 != 0
308     }
309     // not halfway - digit tells all
310     return a.d[nd] >= '5'
311 }
312
313 // Round a to nd digits (or fewer).
314 // If nd is zero, it means we're rounding
315 // just to the left of the digits, as in
316 // 0.09 -> 0.1.
317 func (a *decimal) Round(nd int) {
318     if nd < 0 || nd >= a.nd {
319         return
320     }
321     if shouldRoundUp(a, nd) {
322         a.RoundUp(nd)
323     } else {
324         a.RoundDown(nd)
325     }
326 }
327
328 // Round a down to nd digits (or fewer).
329 func (a *decimal) RoundDown(nd int) {
330     if nd < 0 || nd >= a.nd {
331         return
332     }
333     a.nd = nd
334     trim(a)
335 }
336
337 // Round a up to nd digits (or fewer).
338 func (a *decimal) RoundUp(nd int) {
339     if nd < 0 || nd >= a.nd {
340         return

```

```

341     }
342
343     // round up
344     for i := nd - 1; i >= 0; i-- {
345         c := a.d[i]
346         if c < '9' { // can stop after this digit
347             a.d[i]++
348             a.nd = i + 1
349             return
350         }
351     }
352
353     // Number is all 9s.
354     // Change to single 1 with adjusted decimal point.
355     a.d[0] = '1'
356     a.nd = 1
357     a.dp++
358 }
359
360 // Extract integer part, rounded appropriately.
361 // No guarantees about overflow.
362 func (a *decimal) RoundedInteger() uint64 {
363     if a.dp > 20 {
364         return 0xFFFFFFFFFFFFFFFF
365     }
366     var i int
367     n := uint64(0)
368     for i = 0; i < a.dp && i < a.nd; i++ {
369         n = n*10 + uint64(a.d[i]-'0')
370     }
371     for ; i < a.dp; i++ {
372         n *= 10
373     }
374     if shouldRoundUp(a, a.dp) {
375         n++
376     }
377     return n
378 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strconv/extfloat.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package strconv
6
7 import "math"
8
9 // An extFloat represents an extended floating-point number,
10 // precision than a float64. It does not try to save bits: t
11 // number represented by the structure is mant*(2^exp), with
12 // sign if neg is true.
13 type extFloat struct {
14     mant uint64
15     exp  int
16     neg  bool
17 }
18
19 // Powers of ten taken from double-conversion library.
20 // http://code.google.com/p/double-conversion/
21 const (
22     firstPowerOfTen = -348
23     stepPowerOfTen  = 8
24 )
25
26 var smallPowersOfTen = [...]extFloat{
27     {1 << 63, -63, false}, // 1
28     {0xa << 60, -60, false}, // 1e1
29     {0x64 << 57, -57, false}, // 1e2
30     {0x3e8 << 54, -54, false}, // 1e3
31     {0x2710 << 50, -50, false}, // 1e4
32     {0x186a0 << 47, -47, false}, // 1e5
33     {0xf4240 << 44, -44, false}, // 1e6
34     {0x989680 << 40, -40, false}, // 1e7
35 }
36
37 var powersOfTen = [...]extFloat{
38     {0xfa8fd5a0081c0288, -1220, false}, // 10^-348
39     {0xbaaee17fa23ebf76, -1193, false}, // 10^-340
40     {0x8b16fb203055ac76, -1166, false}, // 10^-332
41     {0xcf42894a5dce35ea, -1140, false}, // 10^-324
42     {0x9a6bb0aa55653b2d, -1113, false}, // 10^-316
43     {0xe61acf033d1a45df, -1087, false}, // 10^-308
44     {0xab70fe17c79ac6ca, -1060, false}, // 10^-300
```

```
45 {0xff77b1fcbebc4f, -1034, false}, // 10^-292
46 {0xbe5691ef416bd60c, -1007, false}, // 10^-284
47 {0x8dd01fad907ffc3c, -980, false}, // 10^-276
48 {0xd3515c2831559a83, -954, false}, // 10^-268
49 {0x9d71ac8fada6c9b5, -927, false}, // 10^-260
50 {0xea9c227723ee8bcb, -901, false}, // 10^-252
51 {0xaecc49914078536d, -874, false}, // 10^-244
52 {0x823c12795db6ce57, -847, false}, // 10^-236
53 {0xc21094364dfb5637, -821, false}, // 10^-228
54 {0x9096ea6f3848984f, -794, false}, // 10^-220
55 {0xd77485cb25823ac7, -768, false}, // 10^-212
56 {0xa086cfc97bf97f4, -741, false}, // 10^-204
57 {0xef340a98172aace5, -715, false}, // 10^-196
58 {0xb23867fb2a35b28e, -688, false}, // 10^-188
59 {0x84c8d4dfd2c63f3b, -661, false}, // 10^-180
60 {0xc5dd44271ad3cdba, -635, false}, // 10^-172
61 {0x936b9fcebb25c996, -608, false}, // 10^-164
62 {0xdbac6c247d62a584, -582, false}, // 10^-156
63 {0xa3ab66580d5fdaf6, -555, false}, // 10^-148
64 {0xf3e2f893dec3f126, -529, false}, // 10^-140
65 {0xb5b5ada8aaff80b8, -502, false}, // 10^-132
66 {0x87625f056c7c4a8b, -475, false}, // 10^-124
67 {0xc9bcff6034c13053, -449, false}, // 10^-116
68 {0x964e858c91ba2655, -422, false}, // 10^-108
69 {0xdff9772470297ebd, -396, false}, // 10^-100
70 {0xa6dfbd9fb8e5b88f, -369, false}, // 10^-92
71 {0xf8a95fcf88747d94, -343, false}, // 10^-84
72 {0xb94470938fa89bcf, -316, false}, // 10^-76
73 {0x8a08f0f8bf0f156b, -289, false}, // 10^-68
74 {0xcdb02555653131b6, -263, false}, // 10^-60
75 {0x993fe2c6d07b7fac, -236, false}, // 10^-52
76 {0xe45c10c42a2b3b06, -210, false}, // 10^-44
77 {0xaa242499697392d3, -183, false}, // 10^-36
78 {0xfd87b5f28300ca0e, -157, false}, // 10^-28
79 {0xbce5086492111aeb, -130, false}, // 10^-20
80 {0x8cbccc096f5088cc, -103, false}, // 10^-12
81 {0xd1b71758e219652c, -77, false}, // 10^-4
82 {0x9c40000000000000, -50, false}, // 10^4
83 {0xe8d4a51000000000, -24, false}, // 10^12
84 {0xad78ebc5ac620000, 3, false}, // 10^20
85 {0x813f3978f8940984, 30, false}, // 10^28
86 {0xc097ce7bc90715b3, 56, false}, // 10^36
87 {0x8f7e32ce7bea5c70, 83, false}, // 10^44
88 {0xd5d238a4abe98068, 109, false}, // 10^52
89 {0x9f4f2726179a2245, 136, false}, // 10^60
90 {0xed63a231d4c4fb27, 162, false}, // 10^68
91 {0xb0de65388cc8ada8, 189, false}, // 10^76
92 {0x83c7088e1aab65db, 216, false}, // 10^84
93 {0xc45d1df942711d9a, 242, false}, // 10^92
94 {0x924d692ca61be758, 269, false}, // 10^100
```

```

95     {0xda01ee641a708dea, 295, false}, // 10^108
96     {0xa26da3999aef774a, 322, false}, // 10^116
97     {0xf209787bb47d6b85, 348, false}, // 10^124
98     {0xb454e4a179dd1877, 375, false}, // 10^132
99     {0x865b86925b9bc5c2, 402, false}, // 10^140
100    {0xc83553c5c8965d3d, 428, false}, // 10^148
101    {0x952ab45cfa97a0b3, 455, false}, // 10^156
102    {0xde469fbd99a05fe3, 481, false}, // 10^164
103    {0xa59bc234db398c25, 508, false}, // 10^172
104    {0xf6c69a72a3989f5c, 534, false}, // 10^180
105    {0xb7dcbf5354e9bece, 561, false}, // 10^188
106    {0x88fcf317f22241e2, 588, false}, // 10^196
107    {0xcc20ce9bd35c78a5, 614, false}, // 10^204
108    {0x98165af37b2153df, 641, false}, // 10^212
109    {0xe2a0b5dc971f303a, 667, false}, // 10^220
110    {0xa8d9d1535ce3b396, 694, false}, // 10^228
111    {0xfb9b7cd9a4a7443c, 720, false}, // 10^236
112    {0xbb764c4ca7a44410, 747, false}, // 10^244
113    {0x8bab8eefb6409c1a, 774, false}, // 10^252
114    {0xd01fef10a657842c, 800, false}, // 10^260
115    {0x9b10a4e5e9913129, 827, false}, // 10^268
116    {0xe7109bfba19c0c9d, 853, false}, // 10^276
117    {0xac2820d9623bf429, 880, false}, // 10^284
118    {0x80444b5e7aa7cf85, 907, false}, // 10^292
119    {0xbf21e44003acdd2d, 933, false}, // 10^300
120    {0x8e679c2f5e44ff8f, 960, false}, // 10^308
121    {0xd433179d9c8cb841, 986, false}, // 10^316
122    {0x9e19db92b4e31ba9, 1013, false}, // 10^324
123    {0xeb96bf6ebadf77d9, 1039, false}, // 10^332
124    {0xaf87023b9bf0ee6b, 1066, false}, // 10^340
125 }
126
127 // floatBits returns the bits of the float64 that best approx
128 // the extFloat passed as receiver. Overflow is set to true
129 // the resulting float64 is ±Inf.
130 func (f *extFloat) floatBits() (bits uint64, overflow bool)
131     flt := &float64info
132     f.Normalize()
133
134     exp := f.exp + 63
135
136     // Exponent too small.
137     if exp < flt.bias+1 {
138         n := flt.bias + 1 - exp
139         f.mant >>= uint(n)
140         exp += n
141     }
142
143     // Extract 1+flt.mantbits bits.

```

```

144     mant := f.mant >> (63 - flt.mantbits)
145     if f.mant&(1<<(62-flt.mantbits)) != 0 {
146         // Round up.
147         mant += 1
148     }
149
150     // Rounding might have added a bit; shift down.
151     if mant == 2<<flt.mantbits {
152         mant >>= 1
153         exp++
154     }
155
156     // Infinities.
157     if exp-flt.bias >= 1<<flt.expbits-1 {
158         goto overflow
159     }
160
161     // Denormalized?
162     if mant&(1<<flt.mantbits) == 0 {
163         exp = flt.bias
164     }
165     goto out
166
167 overflow:
168     // ±Inf
169     mant = 0
170     exp = 1<<flt.expbits - 1 + flt.bias
171     overflow = true
172
173 out:
174     // Assemble bits.
175     bits = mant & (uint64(1)<<flt.mantbits - 1)
176     bits |= uint64((exp-flt.bias)&(1<<flt.expbits-1)) <<
177     if f.neg {
178         bits |= 1 << (flt.mantbits + flt.expbits)
179     }
180     return
181 }
182
183 // Assign sets f to the value of x.
184 func (f *extFloat) Assign(x float64) {
185     if x < 0 {
186         x = -x
187         f.neg = true
188     }
189     x, f.exp = math.Frexp(x)
190     f.mant = uint64(x * float64(1<<64))
191     f.exp -= 64
192 }

```

```

193
194 // AssignComputeBounds sets f to the value of x and returns
195 // lower, upper such that any number in the closed interval
196 // [lower, upper] is converted back to x.
197 func (f *extFloat) AssignComputeBounds(x float64) (lower, up
198 // Special cases.
199 bits := math.Float64bits(x)
200 flt := &float64info
201 neg := bits>>(flt.expbits+flt.mantbits) != 0
202 expBiased := int(bits>>flt.mantbits) & (1<<flt.expbi
203 mant := bits & (uint64(1)<<flt.mantbits - 1)
204
205 if expBiased == 0 {
206     // denormalized.
207     f.mant = mant
208     f.exp = 1 + flt.bias - int(flt.mantbits)
209 } else {
210     f.mant = mant | 1<<flt.mantbits
211     f.exp = expBiased + flt.bias - int(flt.mantb
212 }
213 f.neg = neg
214
215 upper = extFloat{mant: 2*f.mant + 1, exp: f.exp - 1,
216 if mant != 0 || expBiased == 1 {
217     lower = extFloat{mant: 2*f.mant - 1, exp: f.
218 } else {
219     lower = extFloat{mant: 4*f.mant - 1, exp: f.
220 }
221 return
222 }
223
224 // Normalize normalizes f so that the highest bit of the man
225 // set, and returns the number by which the mantissa was lef
226 func (f *extFloat) Normalize() uint {
227     if f.mant == 0 {
228         return 0
229     }
230     exp_before := f.exp
231     for f.mant < (1 << 55) {
232         f.mant <<= 8
233         f.exp -= 8
234     }
235     for f.mant < (1 << 63) {
236         f.mant <<= 1
237         f.exp -= 1
238     }
239     return uint(exp_before - f.exp)
240 }
241
242 // Multiply sets f to the product f*g: the result is correct

```

```

243 // but not normalized.
244 func (f *extFloat) Multiply(g extFloat) {
245     fhi, flo := f.mant>>32, uint64(uint32(f.mant))
246     ghi, glo := g.mant>>32, uint64(uint32(g.mant))
247
248     // Cross products.
249     cross1 := fhi * glo
250     cross2 := flo * ghi
251
252     // f.mant*g.mant is fhi*ghi << 64 + (cross1+cross2)
253     f.mant = fhi*ghi + (cross1 >> 32) + (cross2 >> 32)
254     rem := uint64(uint32(cross1)) + uint64(uint32(cross2)
255     // Round up.
256     rem += (1 << 31)
257
258     f.mant += (rem >> 32)
259     f.exp = f.exp + g.exp + 64
260 }
261
262 var uint64pow10 = [...]uint64{
263     1, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9,
264     1e10, 1e11, 1e12, 1e13, 1e14, 1e15, 1e16, 1e17, 1e18
265 }
266
267 // AssignDecimal sets f to an approximate value of the decir
268 // returns true if the value represented by f is guaranteed
269 // best approximation of d after being rounded to a float64.
270 func (f *extFloat) AssignDecimal(d *decimal) (ok bool) {
271     const uint64digits = 19
272     const errorscale = 8
273     mant10, digits := d.atou64()
274     exp10 := d.dp - digits
275     errors := 0 // An upper bound for error, computed in
276
277     if digits < d.nd {
278         // the decimal number was truncated.
279         errors += errorscale / 2
280     }
281
282     f.mant = mant10
283     f.exp = 0
284     f.neg = d.neg
285
286     // Multiply by powers of ten.
287     i := (exp10 - firstPowerOfTen) / stepPowerOfTen
288     if exp10 < firstPowerOfTen || i >= len(powersOfTen)
289         return false
290     }
291     adjExp := (exp10 - firstPowerOfTen) % stepPowerOfTen

```

```

292
293 // We multiply by exp%step
294 if digits+adjExp <= uint64digits {
295     // We can multiply the mantissa
296     f.mant *= uint64(float64pow10[adjExp])
297     f.Normalize()
298 } else {
299     f.Normalize()
300     f.Multiply(smallPowersOfTen[adjExp])
301     errors += errorscale / 2
302 }
303
304 // We multiply by 10 to the exp - exp%step.
305 f.Multiply(powersOfTen[i])
306 if errors > 0 {
307     errors += 1
308 }
309 errors += errorscale / 2
310
311 // Normalize
312 shift := f.Normalize()
313 errors <<= shift
314
315 // Now f is a good approximation of the decimal.
316 // Check whether the error is too large: that is, if
317 // is perturbed by the error, the resulting float6
318 // The 64 bits mantissa is 1 + 52 bits for float64 +
319 //
320 // In many cases the approximation will be good enou
321 const denormalExp = -1023 - 63
322 flt := &float64info
323 var extrabits uint
324 if f.exp <= denormalExp {
325     extrabits = uint(63 - flt.mantbits + 1 + uin
326 } else {
327     extrabits = uint(63 - flt.mantbits)
328 }
329
330 halfway := uint64(1) << (extrabits - 1)
331 mant_extra := f.mant & (1<<extrabits - 1)
332
333 // Do a signed comparison here! If the error estimat
334 // the mantissa round differently for the conversion
335 // then we can't give a definite answer.
336 if int64(halfway)-int64(errors) < int64(mant_extra)
337     int64(mant_extra) < int64(halfway)+int64(err
338     return false
339 }
340 return true

```

```

341 }
342
343 // Frexp10 is an analogue of math.Frexp for decimal powers.
344 // f by an approximate power of ten 10^-exp, and returns exp
345 // that f*10^exp10 has the same value as the old f, up to an
346 // as well as the index of 10^-exp in the powersOfTen table.
347 // The arguments expMin and expMax constrain the final value
348 // binary exponent of f.
349 func (f *extFloat) frexp10(expMin, expMax int) (exp10, index
350     // it is illegal to call this function with a too re
351     if expMax-expMin <= 25 {
352         panic("strconv: invalid exponent range")
353     }
354     // Find power of ten such that x * 10^n has a binary
355     // between expMin and expMax
356     approxExp10 := -(f.exp + 100) * 28 / 93 // log(10)/1
357     i := (approxExp10 - firstPowerOfTen) / stepPowerOfTe
358 Loop:
359     for {
360         exp := f.exp + powersOfTen[i].exp + 64
361         switch {
362         case exp < expMin:
363             i++
364         case exp > expMax:
365             i--
366         default:
367             break Loop
368         }
369     }
370     // Apply the desired decimal shift on f. It will hav
371     // in the desired range. This is multiplication by 1
372     f.Multiply(powersOfTen[i])
373
374     return -(firstPowerOfTen + i*stepPowerOfTen), i
375 }
376
377 // frexp10Many applies a common shift by a power of ten to a
378 func frexp10Many(expMin, expMax int, a, b, c *extFloat) (exp
379     exp10, i := c.frexp10(expMin, expMax)
380     a.Multiply(powersOfTen[i])
381     b.Multiply(powersOfTen[i])
382     return
383 }
384
385 // ShortestDecimal stores in d the shortest decimal represen
386 // which belongs to the open interval (lower, upper), where
387 // to lie. It returns false whenever the result is unsure. T
388 // uses the Grisu3 algorithm.
389 func (f *extFloat) ShortestDecimal(d *decimal, lower, upper
390     if f.mant == 0 {

```

```

391         d.d[0] = '0'
392         d.nd = 1
393         d.dp = 0
394         d.neg = f.neg
395     }
396     const minExp = -60
397     const maxExp = -32
398     upper.Normalize()
399     // Uniformize exponents.
400     if f.exp > upper.exp {
401         f.mant <<= uint(f.exp - upper.exp)
402         f.exp = upper.exp
403     }
404     if lower.exp > upper.exp {
405         lower.mant <<= uint(lower.exp - upper.exp)
406         lower.exp = upper.exp
407     }
408
409     exp10 := frexp10Many(minExp, maxExp, lower, f, upper
410 // Take a safety margin due to rounding in frexp10Ma
411 upper.mant++
412 lower.mant--
413
414 // The shortest representation of f is either rounde
415 // in any case, it is a truncation of upper.
416 shift := uint(-upper.exp)
417 integer := uint32(upper.mant >> shift)
418 fraction := upper.mant - (uint64(integer) << shift)
419
420 // How far we can go down from upper until the resul
421 allowance := upper.mant - lower.mant
422 // How far we should go to get a very precise result
423 targetDiff := upper.mant - f.mant
424
425 // Count integral digits: there are at most 10.
426 var integerDigits int
427 for i, pow := range uint64pow10 {
428     if uint64(integer) >= pow {
429         integerDigits = i + 1
430     }
431 }
432 for i := 0; i < integerDigits; i++ {
433     pow := uint64pow10[integerDigits-i-1]
434     digit := integer / uint32(pow)
435     d.d[i] = byte(digit + '0')
436     integer -= digit * uint32(pow)
437     // evaluate whether we should stop.
438     if currentDiff := uint64(integer)<<shift + f
439         d.nd = i + 1

```

```

440         d.dp = integerDigits + exp10
441         d.neg = f.neg
442         // Sometimes allowance is so large t
443         // decremented to get closer to f.
444         return adjustLastDigit(d, currentDif
445     }
446 }
447 d.nd = integerDigits
448 d.dp = d.nd + exp10
449 d.neg = f.neg
450
451 // Compute digits of the fractional part. At each st
452 // overflow. The choice of minExp implies that fract
453 var digit int
454 multiplier := uint64(1)
455 for {
456     fraction *= 10
457     multiplier *= 10
458     digit = int(fraction >> shift)
459     d.d[d.nd] = byte(digit + '0')
460     d.nd++
461     fraction -= uint64(digit) << shift
462     if fraction < allowance*multiplier {
463         // We are in the admissible range. N
464         // overflow, that is, allowance > 2^
465         // true due to the limited range of
466         return adjustLastDigit(d,
467             fraction, targetDiff*multipl
468             1<<shift, multiplier*2)
469     }
470 }
471 return false
472 }
473
474 // adjustLastDigit modifies d = x-currentDiff*ε, to get clos
475 // d = x-targetDiff*ε, without becoming smaller than x-maxDi
476 // It assumes that a decimal digit is worth ulpDecimal*ε, an
477 // all data is known with a error estimate of ulpBinary*ε.
478 func adjustLastDigit(d *decimal, currentDiff, targetDiff, ma
479     if ulpDecimal < 2*ulpBinary {
480         // Approximation is too wide.
481         return false
482     }
483     for currentDiff+ulpDecimal/2+ulpBinary < targetDiff
484         d.d[d.nd-1]--
485         currentDiff += ulpDecimal
486     }
487     if currentDiff+ulpDecimal <= targetDiff+ulpDecimal/2
488         // we have two choices, and don't know what

```

```
489             return false
490         }
491         if currentDiff < ulpBinary || currentDiff > maxDiff-
492             // we went too far
493             return false
494     }
495     if d.nd == 1 && d.d[0] == '0' {
496         // the number has actually reached zero.
497         d.nd = 0
498         d.dp = 0
499     }
500     return true
501 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strconv/ftoa.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Binary to decimal floating point conversion.
6 // Algorithm:
7 // 1) store mantissa in multiprecision decimal
8 // 2) shift decimal by exponent
9 // 3) read digits out & format
10
11 package strconv
12
13 import "math"
14
15 // TODO: move elsewhere?
16 type floatInfo struct {
17     mantbits uint
18     expbits  uint
19     bias     int
20 }
21
22 var float32info = floatInfo{23, 8, -127}
23 var float64info = floatInfo{52, 11, -1023}
24
25 // FormatFloat converts the floating-point number f to a str
26 // according to the format fmt and precision prec. It round
27 // result assuming that the original was obtained from a flo
28 // value of bitSize bits (32 for float32, 64 for float64).
29 //
30 // The format fmt is one of
31 // 'b' (-ddddp±ddd, a binary exponent),
32 // 'e' (-d.dddde±dd, a decimal exponent),
33 // 'E' (-d.ddddE±dd, a decimal exponent),
34 // 'f' (-ddd.dddd, no exponent),
35 // 'g' ('e' for large exponents, 'f' otherwise), or
36 // 'G' ('E' for large exponents, 'f' otherwise).
37 //
38 // The precision prec controls the number of digits
39 // (excluding the exponent) printed by the 'e', 'E', 'f', 'g
40 // For 'e', 'E', and 'f' it is the number of digits after th
41 // For 'g' and 'G' it is the total number of digits.
42 // The special precision -1 uses the smallest number of digi
43 // necessary such that ParseFloat will return f exactly.
44 func FormatFloat(f float64, fmt byte, prec, bitSize int) str
```

```

45         return string(genericFtoa(make([]byte, 0), max(prec+4
46     })
47
48     // AppendFloat appends the string form of the floating-point
49     // as generated by FormatFloat, to dst and returns the exten
50     func AppendFloat(dst []byte, f float64, fmt byte, prec int,
51         return genericFtoa(dst, f, fmt, prec, bitSize)
52     }
53
54     func genericFtoa(dst []byte, val float64, fmt byte, prec, bi
55         var bits uint64
56         var flt *floatInfo
57         switch bitSize {
58         case 32:
59             bits = uint64(math.Float32bits(float32(val)))
60             flt = &float32info
61         case 64:
62             bits = math.Float64bits(val)
63             flt = &float64info
64         default:
65             panic("strconv: illegal AppendFloat/FormatFl
66         }
67
68         neg := bits>>(flt.expbits+flt.mantbits) != 0
69         exp := int(bits>>flt.mantbits) & (1<<flt.expbits - 1
70         mant := bits & (uint64(1)<<flt.mantbits - 1)
71
72         switch exp {
73         case 1<<flt.expbits - 1:
74             // Inf, NaN
75             var s string
76             switch {
77             case mant != 0:
78                 s = "NaN"
79             case neg:
80                 s = "-Inf"
81             default:
82                 s = "+Inf"
83             }
84             return append(dst, s...)
85
86         case 0:
87             // denormalized
88             exp++
89
90         default:
91             // add implicit top bit
92             mant |= uint64(1) << flt.mantbits
93         }
94         exp += flt.bias

```

```

95
96 // Pick off easy binary format.
97 if fmt == 'b' {
98     return fmtB(dst, neg, mant, exp, flt)
99 }
100
101 // Negative precision means "only as much as needed
102 shortest := prec < 0
103
104 d := new(decimal)
105 if shortest {
106     ok := false
107     if optimize && bitSize == 64 {
108         // Try Grisu3 algorithm.
109         f := new(extFloat)
110         lower, upper := f.AssignComputeBound
111         ok = f.ShortestDecimal(d, &lower, &u
112     }
113     if !ok {
114         // Create exact decimal representati
115         // The shift is exp - flt.mantbits b
116         // followed by a flt.mantbits fracti
117         // a 1+flt.mantbits-bit integer.
118         d.Assign(mant)
119         d.Shift(exp - int(flt.mantbits))
120         roundShortest(d, mant, exp, flt)
121     }
122     // Precision for shortest representation mod
123     if prec < 0 {
124         switch fmt {
125             case 'e', 'E':
126                 prec = d.nd - 1
127             case 'f':
128                 prec = max(d.nd-d.dp, 0)
129             case 'g', 'G':
130                 prec = d.nd
131         }
132     }
133 } else {
134     // Create exact decimal representation.
135     d.Assign(mant)
136     d.Shift(exp - int(flt.mantbits))
137     // Round appropriately.
138     switch fmt {
139         case 'e', 'E':
140             d.Round(prec + 1)
141         case 'f':
142             d.Round(d.dp + prec)
143         case 'g', 'G':

```

```

144         if prec == 0 {
145             prec = 1
146         }
147         d.Round(prec)
148     }
149 }
150
151 switch fmt {
152 case 'e', 'E':
153     return fmtE(dst, neg, d, prec, fmt)
154 case 'f':
155     return fmtF(dst, neg, d, prec)
156 case 'g', 'G':
157     // trailing fractional zeros in 'e' form will
158     eprec := prec
159     if eprec > d.nd && d.nd >= d.dp {
160         eprec = d.nd
161     }
162     // %e is used if the exponent from the conve
163     // is less than -4 or greater than or equal
164     // if precision was the shortest possible, u
165     if shortest {
166         eprec = 6
167     }
168     exp := d.dp - 1
169     if exp < -4 || exp >= eprec {
170         if prec > d.nd {
171             prec = d.nd
172         }
173         return fmtE(dst, neg, d, prec-1, fmt)
174     }
175     if prec > d.dp {
176         prec = d.nd
177     }
178     return fmtF(dst, neg, d, max(prec-d.dp, 0))
179 }
180
181 // unknown format
182 return append(dst, '%', fmt)
183 }
184
185 // Round d (= mant * 2^exp) to the shortest number of digits
186 // that will let the original floating point value be precis
187 // reconstructed. Size is original floating point size (64
188 func roundShortest(d *decimal, mant uint64, exp int, flt *fl
189     // If mantissa is zero, the number is zero; stop now
190     if mant == 0 {
191         d.nd = 0
192         return

```

```

193     }
194
195     // Compute upper and lower such that any decimal num
196     // between upper and lower (possibly inclusive)
197     // will round to the original floating point number.
198
199     // We may see at once that the number is already sho
200     //
201     // Suppose d is not denormal, so that  $2^{\text{exp}} \leq d < 1$ 
202     // The closest shorter number is at least  $10^{(\text{dp}-\text{nd})}$ 
203     // The lower/upper bounds computed below are at dist
204     // at most  $2^{(\text{exp}-\text{mantbits})}$ .
205     //
206     // So the number is already shortest if  $10^{(\text{dp}-\text{nd})} >$ 
207     // or equivalently  $\log_2(10) * (\text{dp}-\text{nd}) > \text{exp}-\text{mantbits}$ .
208     // It is true if  $332/100 * (\text{dp}-\text{nd}) \geq \text{exp}-\text{mantbits}$  (lo
209     minexp := flt.bias + 1 // minimum possible exponent
210     if exp > minexp && 332*(d.dp-d.nd) >= 100*(exp-int(f
211         // The number is already shortest.
212         return
213     }
214
215     // d = mant << (exp - mantbits)
216     // Next highest floating point number is mant+1 << e
217     // Our upper bound is halfway inbetween, mant*2+1 <<
218     upper := new(decimal)
219     upper.Assign(mant*2 + 1)
220     upper.Shift(exp - int(flt.mantbits) - 1)
221
222     // d = mant << (exp - mantbits)
223     // Next lowest floating point number is mant-1 << ex
224     // unless mant-1 drops the significant bit and exp i
225     // in which case the next lowest is mant*2-1 << exp-
226     // Either way, call it mantlo << explo-mantbits.
227     // Our lower bound is halfway inbetween, mantlo*2+1
228     var mantlo uint64
229     var explo int
230     if mant > 1<<flt.mantbits || exp == minexp {
231         mantlo = mant - 1
232         explo = exp
233     } else {
234         mantlo = mant*2 - 1
235         explo = exp - 1
236     }
237     lower := new(decimal)
238     lower.Assign(mantlo*2 + 1)
239     lower.Shift(explo - int(flt.mantbits) - 1)
240
241     // The upper and lower bounds are possible outputs o
242     // the original mantissa is even, so that IEEE round

```

```

243 // would round to the original mantissa and not the
244 inclusive := mant%2 == 0
245
246 // Now we can figure out the minimum number of digit
247 // Walk along until d has distinguished itself from
248 for i := 0; i < d.nd; i++ {
249     var l, m, u byte // lower, middle, upper dig
250     if i < lower.nd {
251         l = lower.d[i]
252     } else {
253         l = '0'
254     }
255     m = d.d[i]
256     if i < upper.nd {
257         u = upper.d[i]
258     } else {
259         u = '0'
260     }
261
262     // Okay to round down (truncate) if lower ha
263     // or if lower is inclusive and is exactly t
264     okdown := l != m || (inclusive && l == m &&
265
266     // Okay to round up if upper has a different
267     // either upper is inclusive or upper is big
268     okup := m != u && (inclusive || m+1 < u || i
269
270     // If it's okay to do either, then round to
271     // If it's okay to do only one, do it.
272     switch {
273     case okdown && okup:
274         d.Round(i + 1)
275         return
276     case okdown:
277         d.RoundDown(i + 1)
278         return
279     case okup:
280         d.RoundUp(i + 1)
281         return
282     }
283 }
284 }
285
286 // %e: -d.ddddde±dd
287 func fmtE(dst []byte, neg bool, d *decimal, prec int, fmt by
288     // sign
289     if neg {
290         dst = append(dst, '-')
291     }

```

```

292
293 // first digit
294 ch := byte('0')
295 if d.nd != 0 {
296     ch = d.d[0]
297 }
298 dst = append(dst, ch)
299
300 // .moredigits
301 if prec > 0 {
302     dst = append(dst, '.')
303     for i := 1; i <= prec; i++ {
304         ch = '0'
305         if i < d.nd {
306             ch = d.d[i]
307         }
308         dst = append(dst, ch)
309     }
310 }
311
312 // e±
313 dst = append(dst, fmt)
314 exp := d.dp - 1
315 if d.nd == 0 { // special case: 0 has exponent 0
316     exp = 0
317 }
318 if exp < 0 {
319     ch = '-'
320     exp = -exp
321 } else {
322     ch = '+'
323 }
324 dst = append(dst, ch)
325
326 // dddd
327 var buf [3]byte
328 i := len(buf)
329 for exp >= 10 {
330     i--
331     buf[i] = byte(exp%10 + '0')
332     exp /= 10
333 }
334 // exp < 10
335 i--
336 buf[i] = byte(exp + '0')
337
338 // leading zeroes
339 if i > len(buf)-2 {
340     i--

```

```

341         buf[i] = '0'
342     }
343
344     return append(dst, buf[i:]...)
345 }
346
347 // %f: -ddddddd.ddddd
348 func fmtF(dst []byte, neg bool, d *decimal, prec int) []byte
349     // sign
350     if neg {
351         dst = append(dst, '-')
352     }
353
354     // integer, padded with zeros as needed.
355     if d.dp > 0 {
356         var i int
357         for i = 0; i < d.dp && i < d.nd; i++ {
358             dst = append(dst, d.d[i])
359         }
360         for ; i < d.dp; i++ {
361             dst = append(dst, '0')
362         }
363     } else {
364         dst = append(dst, '0')
365     }
366
367     // fraction
368     if prec > 0 {
369         dst = append(dst, '.')
370         for i := 0; i < prec; i++ {
371             ch := byte('0')
372             if j := d.dp + i; 0 <= j && j < d.nd
373                 ch = d.d[j]
374         }
375         dst = append(dst, ch)
376     }
377 }
378
379     return dst
380 }
381
382 // %b: -dddddddp+ddd
383 func fmtB(dst []byte, neg bool, mant uint64, exp int, flt *f
384     var buf [50]byte
385     w := len(buf)
386     exp -= int(flt.mantbits)
387     esign := byte('+')
388     if exp < 0 {
389         esign = '-'
390         exp = -exp

```

```

391     }
392     n := 0
393     for exp > 0 || n < 1 {
394         n++
395         w--
396         buf[w] = byte(exp%10 + '0')
397         exp /= 10
398     }
399     w--
400     buf[w] = esign
401     w--
402     buf[w] = 'p'
403     n = 0
404     for mant > 0 || n < 1 {
405         n++
406         w--
407         buf[w] = byte(mant%10 + '0')
408         mant /= 10
409     }
410     if neg {
411         w--
412         buf[w] = '-'
413     }
414     return append(dst, buf[w:]...)
415 }
416
417 func max(a, b int) int {
418     if a > b {
419         return a
420     }
421     return b
422 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strconv/isprint.go

```
1 // DO NOT EDIT. GENERATED BY
2 //     go run makeisprint.go >x && mv x isprint.go
3
4 package strconv
5
6 // (474+134+42)*2 + (180)*4 = 2020 bytes
7
8 var isPrint16 = []uint16{
9     0x0020, 0x007e,
10    0x00a1, 0x0377,
11    0x037a, 0x037e,
12    0x0384, 0x0527,
13    0x0531, 0x0556,
14    0x0559, 0x058a,
15    0x0591, 0x05c7,
16    0x05d0, 0x05ea,
17    0x05f0, 0x05f4,
18    0x0606, 0x061b,
19    0x061e, 0x070d,
20    0x0710, 0x074a,
21    0x074d, 0x07b1,
22    0x07c0, 0x07fa,
23    0x0800, 0x082d,
24    0x0830, 0x085b,
25    0x085e, 0x085e,
26    0x0900, 0x098c,
27    0x098f, 0x0990,
28    0x0993, 0x09b2,
29    0x09b6, 0x09b9,
30    0x09bc, 0x09c4,
31    0x09c7, 0x09c8,
32    0x09cb, 0x09ce,
33    0x09d7, 0x09d7,
34    0x09dc, 0x09e3,
35    0x09e6, 0x09fb,
36    0x0a01, 0x0a0a,
37    0x0a0f, 0x0a10,
38    0x0a13, 0x0a39,
39    0x0a3c, 0x0a42,
40    0x0a47, 0x0a48,
41    0x0a4b, 0x0a4d,
42    0x0a51, 0x0a51,
43    0x0a59, 0x0a5e,
44    0x0a66, 0x0a75,
```

45	0x0a81,	0x0ab9,
46	0x0abc,	0x0acd,
47	0x0ad0,	0x0ad0,
48	0x0ae0,	0x0ae3,
49	0x0ae6,	0x0af1,
50	0x0b01,	0x0b0c,
51	0x0b0f,	0x0b10,
52	0x0b13,	0x0b39,
53	0x0b3c,	0x0b44,
54	0x0b47,	0x0b48,
55	0x0b4b,	0x0b4d,
56	0x0b56,	0x0b57,
57	0x0b5c,	0x0b63,
58	0x0b66,	0x0b77,
59	0x0b82,	0x0b8a,
60	0x0b8e,	0x0b95,
61	0x0b99,	0x0b9f,
62	0x0ba3,	0x0ba4,
63	0x0ba8,	0x0baa,
64	0x0bae,	0x0bb9,
65	0x0bbe,	0x0bc2,
66	0x0bc6,	0x0bcd,
67	0x0bd0,	0x0bd0,
68	0x0bd7,	0x0bd7,
69	0x0be6,	0x0bfa,
70	0x0c01,	0x0c39,
71	0x0c3d,	0x0c4d,
72	0x0c55,	0x0c59,
73	0x0c60,	0x0c63,
74	0x0c66,	0x0c6f,
75	0x0c78,	0x0c7f,
76	0x0c82,	0x0cb9,
77	0x0cbc,	0x0ccd,
78	0x0cd5,	0x0cd6,
79	0x0cde,	0x0ce3,
80	0x0ce6,	0x0cf2,
81	0x0d02,	0x0d3a,
82	0x0d3d,	0x0d4e,
83	0x0d57,	0x0d57,
84	0x0d60,	0x0d63,
85	0x0d66,	0x0d75,
86	0x0d79,	0x0d7f,
87	0x0d82,	0x0d96,
88	0x0d9a,	0x0dbd,
89	0x0dc0,	0x0dc6,
90	0x0dca,	0x0dca,
91	0x0dcf,	0x0ddf,
92	0x0df2,	0x0df4,
93	0x0e01,	0x0e3a,
94	0x0e3f,	0x0e5b,

95	0x0e81,	0x0e84,
96	0x0e87,	0x0e8a,
97	0x0e8d,	0x0e8d,
98	0x0e94,	0x0ea7,
99	0x0eaa,	0x0ebd,
100	0x0ec0,	0x0ecd,
101	0x0ed0,	0x0ed9,
102	0x0edc,	0x0edd,
103	0x0f00,	0x0f6c,
104	0x0f71,	0x0fda,
105	0x1000,	0x10c5,
106	0x10d0,	0x10fc,
107	0x1100,	0x124d,
108	0x1250,	0x125d,
109	0x1260,	0x128d,
110	0x1290,	0x12b5,
111	0x12b8,	0x12c5,
112	0x12c8,	0x1315,
113	0x1318,	0x135a,
114	0x135d,	0x137c,
115	0x1380,	0x1399,
116	0x13a0,	0x13f4,
117	0x1400,	0x169c,
118	0x16a0,	0x16f0,
119	0x1700,	0x1714,
120	0x1720,	0x1736,
121	0x1740,	0x1753,
122	0x1760,	0x1773,
123	0x1780,	0x17b3,
124	0x17b6,	0x17dd,
125	0x17e0,	0x17e9,
126	0x17f0,	0x17f9,
127	0x1800,	0x180d,
128	0x1810,	0x1819,
129	0x1820,	0x1877,
130	0x1880,	0x18aa,
131	0x18b0,	0x18f5,
132	0x1900,	0x191c,
133	0x1920,	0x192b,
134	0x1930,	0x193b,
135	0x1940,	0x1940,
136	0x1944,	0x196d,
137	0x1970,	0x1974,
138	0x1980,	0x19ab,
139	0x19b0,	0x19c9,
140	0x19d0,	0x19da,
141	0x19de,	0x1a1b,
142	0x1a1e,	0x1a7c,
143	0x1a7f,	0x1a89,

144	0x1a90,	0x1a99,
145	0x1aa0,	0x1aad,
146	0x1b00,	0x1b4b,
147	0x1b50,	0x1b7c,
148	0x1b80,	0x1baa,
149	0x1bae,	0x1bb9,
150	0x1bc0,	0x1bf3,
151	0x1bfc,	0x1c37,
152	0x1c3b,	0x1c49,
153	0x1c4d,	0x1c7f,
154	0x1cd0,	0x1cf2,
155	0x1d00,	0x1de6,
156	0x1dfc,	0x1f15,
157	0x1f18,	0x1f1d,
158	0x1f20,	0x1f45,
159	0x1f48,	0x1f4d,
160	0x1f50,	0x1f7d,
161	0x1f80,	0x1fd3,
162	0x1fd6,	0x1fef,
163	0x1ff2,	0x1ffe,
164	0x2010,	0x2027,
165	0x2030,	0x205e,
166	0x2070,	0x2071,
167	0x2074,	0x209c,
168	0x20a0,	0x20b9,
169	0x20d0,	0x20f0,
170	0x2100,	0x2189,
171	0x2190,	0x23f3,
172	0x2400,	0x2426,
173	0x2440,	0x244a,
174	0x2460,	0x2b4c,
175	0x2b50,	0x2b59,
176	0x2c00,	0x2cf1,
177	0x2cf9,	0x2d25,
178	0x2d30,	0x2d65,
179	0x2d6f,	0x2d70,
180	0x2d7f,	0x2d96,
181	0x2da0,	0x2e31,
182	0x2e80,	0x2ef3,
183	0x2f00,	0x2fd5,
184	0x2ff0,	0x2ffb,
185	0x3001,	0x3096,
186	0x3099,	0x30ff,
187	0x3105,	0x312d,
188	0x3131,	0x31ba,
189	0x31c0,	0x31e3,
190	0x31f0,	0x4db5,
191	0x4dc0,	0x9fcb,
192	0xa000,	0xa48c,

193	0xa490,	0xa4c6,
194	0xa4d0,	0xa62b,
195	0xa640,	0xa673,
196	0xa67c,	0xa697,
197	0xa6a0,	0xa6f7,
198	0xa700,	0xa791,
199	0xa7a0,	0xa7a9,
200	0xa7fa,	0xa82b,
201	0xa830,	0xa839,
202	0xa840,	0xa877,
203	0xa880,	0xa8c4,
204	0xa8ce,	0xa8d9,
205	0xa8e0,	0xa8fb,
206	0xa900,	0xa953,
207	0xa95f,	0xa97c,
208	0xa980,	0xa9d9,
209	0xa9de,	0xa9df,
210	0xaa00,	0xaa36,
211	0xaa40,	0xaa4d,
212	0xaa50,	0xaa59,
213	0xaa5c,	0xaa7b,
214	0xaa80,	0xaac2,
215	0xaadb,	0xaadf,
216	0xab01,	0xab06,
217	0xab09,	0xab0e,
218	0xab11,	0xab16,
219	0xab20,	0xab2e,
220	0xabc0,	0xabed,
221	0xabf0,	0xabf9,
222	0xac00,	0xd7a3,
223	0xd7b0,	0xd7c6,
224	0xd7cb,	0xd7fb,
225	0xf900,	0xfa2d,
226	0xfa30,	0xfa6d,
227	0xfa70,	0xfad9,
228	0xfb00,	0xfb06,
229	0xfb13,	0xfb17,
230	0xfb1d,	0xfbc1,
231	0xfb3d,	0xfd3f,
232	0xfd50,	0xfd8f,
233	0xfd92,	0xfdc7,
234	0xfdf0,	0xfdfd,
235	0xfe00,	0xfe19,
236	0xfe20,	0xfe26,
237	0xfe30,	0xfe6b,
238	0xfe70,	0xfefc,
239	0xff01,	0xffbe,
240	0xffc2,	0xffc7,
241	0xffca,	0xffcf,
242	0xffd2,	0xffd7,

```
243         0xffda, 0xffdc,
244         0xffe0, 0xffee,
245         0xffffc, 0xffffd,
246     }
247
248     var isNotPrint16 = []uint16{
249         0x00ad,
250         0x038b,
251         0x038d,
252         0x03a2,
253         0x0560,
254         0x0588,
255         0x06dd,
256         0x083f,
257         0x0978,
258         0x0980,
259         0x0984,
260         0x09a9,
261         0x09b1,
262         0x09de,
263         0x0a04,
264         0x0a29,
265         0x0a31,
266         0x0a34,
267         0x0a37,
268         0x0a3d,
269         0x0a5d,
270         0x0a84,
271         0x0a8e,
272         0x0a92,
273         0x0aa9,
274         0x0ab1,
275         0x0ab4,
276         0x0ac6,
277         0x0aca,
278         0x0af0,
279         0x0b04,
280         0x0b29,
281         0x0b31,
282         0x0b34,
283         0x0b5e,
284         0x0b84,
285         0x0b91,
286         0x0b9b,
287         0x0b9d,
288         0x0bc9,
289         0x0c04,
290         0x0c0d,
291         0x0c11,
```

292	0x0c29,
293	0x0c34,
294	0x0c45,
295	0x0c49,
296	0x0c57,
297	0x0c84,
298	0x0c8d,
299	0x0c91,
300	0x0ca9,
301	0x0cb4,
302	0x0cc5,
303	0x0cc9,
304	0x0cdf,
305	0x0cf0,
306	0x0d04,
307	0x0d0d,
308	0x0d11,
309	0x0d45,
310	0x0d49,
311	0x0d84,
312	0x0db2,
313	0x0dbc,
314	0x0dd5,
315	0x0dd7,
316	0x0e83,
317	0x0e89,
318	0x0e98,
319	0x0ea0,
320	0x0ea4,
321	0x0ea6,
322	0x0eac,
323	0x0eba,
324	0x0ec5,
325	0x0ec7,
326	0x0f48,
327	0x0f98,
328	0x0fbd,
329	0x0fcd,
330	0x1249,
331	0x1257,
332	0x1259,
333	0x1289,
334	0x12b1,
335	0x12bf,
336	0x12c1,
337	0x12d7,
338	0x1311,
339	0x1680,
340	0x170d,

```
341         0x176d,  
342         0x1771,  
343         0x1a5f,  
344         0x1f58,  
345         0x1f5a,  
346         0x1f5c,  
347         0x1f5e,  
348         0x1fb5,  
349         0x1fc5,  
350         0x1fdc,  
351         0x1ff5,  
352         0x208f,  
353         0x2700,  
354         0x27cb,  
355         0x27cd,  
356         0x2c2f,  
357         0x2c5f,  
358         0x2da7,  
359         0x2daf,  
360         0x2db7,  
361         0x2dbf,  
362         0x2dc7,  
363         0x2dcf,  
364         0x2dd7,  
365         0x2ddf,  
366         0x2e9a,  
367         0x3040,  
368         0x318f,  
369         0x321f,  
370         0x32ff,  
371         0xa78f,  
372         0xa9ce,  
373         0xab27,  
374         0xfb37,  
375         0xfb3d,  
376         0xfb3f,  
377         0xfb42,  
378         0xfb45,  
379         0xfe53,  
380         0xfe67,  
381         0xfe75,  
382         0xffe7,  
383     }  
384  
385     var isPrint32 = []uint32{  
386         0x010000, 0x01004d,  
387         0x010050, 0x01005d,  
388         0x010080, 0x0100fa,  
389         0x010100, 0x010102,  
390         0x010107, 0x010133,
```

391	0x010137,	0x01018a,
392	0x010190,	0x01019b,
393	0x0101d0,	0x0101fd,
394	0x010280,	0x01029c,
395	0x0102a0,	0x0102d0,
396	0x010300,	0x010323,
397	0x010330,	0x01034a,
398	0x010380,	0x0103c3,
399	0x0103c8,	0x0103d5,
400	0x010400,	0x01049d,
401	0x0104a0,	0x0104a9,
402	0x010800,	0x010805,
403	0x010808,	0x010838,
404	0x01083c,	0x01083c,
405	0x01083f,	0x01085f,
406	0x010900,	0x01091b,
407	0x01091f,	0x010939,
408	0x01093f,	0x01093f,
409	0x010a00,	0x010a06,
410	0x010a0c,	0x010a33,
411	0x010a38,	0x010a3a,
412	0x010a3f,	0x010a47,
413	0x010a50,	0x010a58,
414	0x010a60,	0x010a7f,
415	0x010b00,	0x010b35,
416	0x010b39,	0x010b55,
417	0x010b58,	0x010b72,
418	0x010b78,	0x010b7f,
419	0x010c00,	0x010c48,
420	0x010e60,	0x010e7e,
421	0x011000,	0x01104d,
422	0x011052,	0x01106f,
423	0x011080,	0x0110c1,
424	0x012000,	0x01236e,
425	0x012400,	0x012462,
426	0x012470,	0x012473,
427	0x013000,	0x01342e,
428	0x016800,	0x016a38,
429	0x01b000,	0x01b001,
430	0x01d000,	0x01d0f5,
431	0x01d100,	0x01d126,
432	0x01d129,	0x01d172,
433	0x01d17b,	0x01d1dd,
434	0x01d200,	0x01d245,
435	0x01d300,	0x01d356,
436	0x01d360,	0x01d371,
437	0x01d400,	0x01d49f,
438	0x01d4a2,	0x01d4a2,
439	0x01d4a5,	0x01d4a6,

```
440         0x01d4a9, 0x01d50a,  
441         0x01d50d, 0x01d546,  
442         0x01d54a, 0x01d6a5,  
443         0x01d6a8, 0x01d7cb,  
444         0x01d7ce, 0x01d7ff,  
445         0x01f000, 0x01f02b,  
446         0x01f030, 0x01f093,  
447         0x01f0a0, 0x01f0ae,  
448         0x01f0b1, 0x01f0be,  
449         0x01f0c1, 0x01f0df,  
450         0x01f100, 0x01f10a,  
451         0x01f110, 0x01f169,  
452         0x01f170, 0x01f19a,  
453         0x01f1e6, 0x01f202,  
454         0x01f210, 0x01f23a,  
455         0x01f240, 0x01f248,  
456         0x01f250, 0x01f251,  
457         0x01f300, 0x01f320,  
458         0x01f330, 0x01f37c,  
459         0x01f380, 0x01f393,  
460         0x01f3a0, 0x01f3ca,  
461         0x01f3e0, 0x01f3f0,  
462         0x01f400, 0x01f4fc,  
463         0x01f500, 0x01f53d,  
464         0x01f550, 0x01f567,  
465         0x01f5fb, 0x01f625,  
466         0x01f628, 0x01f62d,  
467         0x01f630, 0x01f640,  
468         0x01f645, 0x01f64f,  
469         0x01f680, 0x01f6c5,  
470         0x01f700, 0x01f773,  
471         0x020000, 0x02a6d6,  
472         0x02a700, 0x02b734,  
473         0x02b740, 0x02b81d,  
474         0x02f800, 0x02fa1d,  
475         0x0e0100, 0x0e01ef,  
476     }  
477  
478     var isNotPrint32 = []uint16{ // add 0x10000 to each entry  
479         0x000c,  
480         0x0027,  
481         0x003b,  
482         0x003e,  
483         0x031f,  
484         0x039e,  
485         0x0809,  
486         0x0836,  
487         0x0856,  
488         0x0a04,
```

```
489         0x0a14,  
490         0x0a18,  
491         0x10bd,  
492         0xd455,  
493         0xd49d,  
494         0xd4ad,  
495         0xd4ba,  
496         0xd4bc,  
497         0xd4c4,  
498         0xd506,  
499         0xd515,  
500         0xd51d,  
501         0xd53a,  
502         0xd53f,  
503         0xd545,  
504         0xd551,  
505         0xf0d0,  
506         0xf12f,  
507         0xf336,  
508         0xf3c5,  
509         0xf43f,  
510         0xf441,  
511         0xf4f8,  
512         0xf600,  
513         0xf611,  
514         0xf615,  
515         0xf617,  
516         0xf619,  
517         0xf61b,  
518         0xf61f,  
519         0xf62c,  
520         0xf634,  
521     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strconv/itoa.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package strconv
6
7 // FormatUint returns the string representation of i in the
8 func FormatUint(i uint64, base int) string {
9     _, s := formatBits(nil, i, base, false, false)
10    return s
11 }
12
13 // FormatInt returns the string representation of i in the g
14 func FormatInt(i int64, base int) string {
15     _, s := formatBits(nil, uint64(i), base, i < 0, fals
16    return s
17 }
18
19 // Itoa is shorthand for FormatInt(i, 10).
20 func Itoa(i int) string {
21    return FormatInt(int64(i), 10)
22 }
23
24 // AppendInt appends the string form of the integer i,
25 // as generated by FormatInt, to dst and returns the extende
26 func AppendInt(dst []byte, i int64, base int) []byte {
27    dst, _ = formatBits(dst, uint64(i), base, i < 0, tru
28    return dst
29 }
30
31 // AppendUint appends the string form of the unsigned intege
32 // as generated by FormatUint, to dst and returns the extend
33 func AppendUint(dst []byte, i uint64, base int) []byte {
34    dst, _ = formatBits(dst, i, base, false, true)
35    return dst
36 }
37
38 const (
39    digits    = "0123456789abcdefghijklmnopqrstuvwxyz"
40    digits01  = "0123456789012345678901234567890123456789
41    digits10  = "0000000000111111111122222222223333333333
42 )
43
44 var shifts = [len(digits) + 1]uint{
```

```

45         1 << 1: 1,
46         1 << 2: 2,
47         1 << 3: 3,
48         1 << 4: 4,
49         1 << 5: 5,
50     }
51
52     // formatBits computes the string representation of u in the
53     // If neg is set, u is treated as negative int64 value. If a
54     // set, the string is appended to dst and the resulting byte
55     // returned as the first result value; otherwise the string
56     // as the second result value.
57     //
58     func formatBits(dst []byte, u uint64, base int, neg, append_
59         if base < 2 || base > len(digits) {
60             panic("strconv: illegal AppendInt/FormatInt
61         }
62         // 2 <= base && base <= len(digits)
63
64         var a [64 + 1]byte // +1 for sign of 64bit value in
65         i := len(a)
66
67         if neg {
68             u = -u
69         }
70
71         // convert bits
72         if base == 10 {
73             // common case: use constants for / and % be
74             // the compiler can optimize it into a multi
75             // and unroll loop
76             for u >= 100 {
77                 i -= 2
78                 q := u / 100
79                 j := uintptr(u - q*100)
80                 a[i+1] = digits01[j]
81                 a[i+0] = digits10[j]
82                 u = q
83             }
84             if u >= 10 {
85                 i--
86                 q := u / 10
87                 a[i] = digits[uintptr(u-q*10)]
88                 u = q
89             }
90
91         } else if s := shifts[base]; s > 0 {
92             // base is power of 2: use shifts and masks
93             b := uint64(base)
94             m := uintptr(b) - 1 // == 1<<s - 1

```

```

95         for u >= b {
96             i--
97             a[i] = digits[uintptr(u)&m]
98             u >>= s
99         }
100
101     } else {
102         // general case
103         b := uint64(base)
104         for u >= b {
105             i--
106             a[i] = digits[uintptr(u%b)]
107             u /= b
108         }
109     }
110
111     // u < base
112     i--
113     a[i] = digits[uintptr(u)]
114
115     // add sign, if any
116     if neg {
117         i--
118         a[i] = '-'
119     }
120
121     if append_ {
122         d = append(dst, a[i:]...)
123         return
124     }
125     s = string(a[i:])
126     return
127 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strconv/quote.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package strconv
6
7 import (
8     "unicode/utf8"
9 )
10
11 const lowerhex = "0123456789abcdef"
12
13 func quotewith(s string, quote byte, ASCIIonly bool) string
14     var runeTmp [utf8.UTFMax]byte
15     buf := make([]byte, 0, 3*len(s)/2) // Try to avoid m
16     buf = append(buf, quote)
17     for width := 0; len(s) > 0; s = s[width:] {
18         r := rune(s[0])
19         width = 1
20         if r >= utf8.RuneSelf {
21             r, width = utf8.DecodeRuneInString(s)
22         }
23         if width == 1 && r == utf8.RuneError {
24             buf = append(buf, `\\x`...)
25             buf = append(buf, lowerhex[s[0]>>4])
26             buf = append(buf, lowerhex[s[0]&0xF])
27             continue
28         }
29         if r == rune(quote) || r == `\\` { // always
30             buf = append(buf, `\\`)
31             buf = append(buf, byte(r))
32             continue
33         }
34         if ASCIIonly {
35             if r < utf8.RuneSelf && IsPrint(r) {
36                 buf = append(buf, byte(r))
37                 continue
38             }
39         } else if IsPrint(r) {
40             n := utf8.EncodeRune(runeTmp[:], r)
41             buf = append(buf, runeTmp[:n]...)
42             continue
43         }
44         switch r {
```

```

45         case '\a':
46             buf = append(buf, `\\a`...)
47         case '\b':
48             buf = append(buf, `\\b`...)
49         case '\f':
50             buf = append(buf, `\\f`...)
51         case '\n':
52             buf = append(buf, `\\n`...)
53         case '\r':
54             buf = append(buf, `\\r`...)
55         case '\t':
56             buf = append(buf, `\\t`...)
57         case '\\v':
58             buf = append(buf, `\\v`...)
59         default:
60             switch {
61             case r < ' ':
62                 buf = append(buf, `\\x`...)
63                 buf = append(buf, lowerhex[s])
64                 buf = append(buf, lowerhex[s])
65             case r > utf8.MaxRune:
66                 r = 0xFFFF
67                 fallthrough
68             case r < 0x10000:
69                 buf = append(buf, `\\u`...)
70                 for s := 12; s >= 0; s -= 4
71                     buf = append(buf, lo
72                 }
73             default:
74                 buf = append(buf, `\\U`...)
75                 for s := 28; s >= 0; s -= 4
76                     buf = append(buf, lo
77                 }
78             }
79         }
80     }
81     buf = append(buf, quote)
82     return string(buf)
83 }
84 }
85
86 // Quote returns a double-quoted Go string literal represent
87 // returned string uses Go escape sequences (\t, \n, \xFF, \
88 // control characters and non-printable characters as define
89 // IsPrint.
90 func Quote(s string) string {
91     return quoteWith(s, '"', false)
92 }
93
94 // AppendQuote appends a double-quoted Go string literal rep

```

```

95 // as generated by Quote, to dst and returns the extended bu
96 func AppendQuote(dst []byte, s string) []byte {
97     return append(dst, Quote(s)...)
98 }
99
100 // QuoteToASCII returns a double-quoted Go string literal re
101 // The returned string uses Go escape sequences (\t, \n, \xF
102 // non-ASCII characters and non-printable characters as defi
103 func QuoteToASCII(s string) string {
104     return quoteWith(s, '"', true)
105 }
106
107 // AppendQuoteToASCII appends a double-quoted Go string lite
108 // as generated by QuoteToASCII, to dst and returns the exte
109 func AppendQuoteToASCII(dst []byte, s string) []byte {
110     return append(dst, QuoteToASCII(s)...)
111 }
112
113 // QuoteRune returns a single-quoted Go character literal re
114 // rune. The returned string uses Go escape sequences (\t,
115 // for control characters and non-printable characters as de
116 func QuoteRune(r rune) string {
117     // TODO: avoid the allocation here.
118     return quoteWith(string(r), '\'', false)
119 }
120
121 // AppendQuoteRune appends a single-quoted Go character lite
122 // as generated by QuoteRune, to dst and returns the extende
123 func AppendQuoteRune(dst []byte, r rune) []byte {
124     return append(dst, QuoteRune(r)...)
125 }
126
127 // QuoteRuneToASCII returns a single-quoted Go character lit
128 // the rune. The returned string uses Go escape sequences (
129 // \u0100) for non-ASCII characters and non-printable charac
130 // by IsPrint.
131 func QuoteRuneToASCII(r rune) string {
132     // TODO: avoid the allocation here.
133     return quoteWith(string(r), '\'', true)
134 }
135
136 // AppendQuoteRune appends a single-quoted Go character lite
137 // as generated by QuoteRuneToASCII, to dst and returns the
138 func AppendQuoteRuneToASCII(dst []byte, r rune) []byte {
139     return append(dst, QuoteRuneToASCII(r)...)
140 }
141
142 // CanBackquote returns whether the string s would be
143 // a valid Go string literal if enclosed in backquotes.

```

```

144 func CanBackquote(s string) bool {
145     for i := 0; i < len(s); i++ {
146         if (s[i] < ' ' && s[i] != '\t') || s[i] == '
147             return false
148         }
149     }
150     return true
151 }
152
153 func unhex(b byte) (v rune, ok bool) {
154     c := rune(b)
155     switch {
156     case '0' <= c && c <= '9':
157         return c - '0', true
158     case 'a' <= c && c <= 'f':
159         return c - 'a' + 10, true
160     case 'A' <= c && c <= 'F':
161         return c - 'A' + 10, true
162     }
163     return
164 }
165
166 // UnquoteChar decodes the first character or byte in the es
167 // or character literal represented by the string s.
168 // It returns four values:
169 //
170 //     1) value, the decoded Unicode code point or byte val
171 //     2) multibyte, a boolean indicating whether the decod
172 //     3) tail, the remainder of the string after the chara
173 //     4) an error that will be nil if the character is syn
174 //
175 // The second argument, quote, specifies the type of literal
176 // and therefore which escaped quote character is permitted.
177 // If set to a single quote, it permits the sequence \' and
178 // If set to a double quote, it permits \" and disallows une
179 // If set to zero, it does not permit either escape and allo
180 func UnquoteChar(s string, quote byte) (value rune, multibyt
181     // easy cases
182     switch c := s[0]; {
183     case c == quote && (quote == '\\' || quote == '"'):
184         err = ErrSyntax
185         return
186     case c >= utf8.RuneSelf:
187         r, size := utf8.DecodeRuneInString(s)
188         return r, true, s[size:], nil
189     case c != '\\':
190         return rune(s[0]), false, s[1:], nil
191     }
192

```

```

193 // hard case: c is backslash
194 if len(s) <= 1 {
195     err = ErrSyntax
196     return
197 }
198 c := s[1]
199 s = s[2:]
200
201 switch c {
202 case 'a':
203     value = '\a'
204 case 'b':
205     value = '\b'
206 case 'f':
207     value = '\f'
208 case 'n':
209     value = '\n'
210 case 'r':
211     value = '\r'
212 case 't':
213     value = '\t'
214 case 'v':
215     value = '\v'
216 case 'x', 'u', 'U':
217     n := 0
218     switch c {
219     case 'x':
220         n = 2
221     case 'u':
222         n = 4
223     case 'U':
224         n = 8
225     }
226     var v rune
227     if len(s) < n {
228         err = ErrSyntax
229         return
230     }
231     for j := 0; j < n; j++ {
232         x, ok := unhex(s[j])
233         if !ok {
234             err = ErrSyntax
235             return
236         }
237         v = v<<4 | x
238     }
239     s = s[n:]
240     if c == 'x' {
241         // single-byte string, possibly not
242         value = v

```

```

243         break
244     }
245     if v > utf8.MaxRune {
246         err = ErrSyntax
247         return
248     }
249     value = v
250     multibyte = true
251     case '0', '1', '2', '3', '4', '5', '6', '7':
252         v := rune(c) - '0'
253         if len(s) < 2 {
254             err = ErrSyntax
255             return
256         }
257         for j := 0; j < 2; j++ { // one digit ahead
258             x := rune(s[j]) - '0'
259             if x < 0 || x > 7 {
260                 err = ErrSyntax
261                 return
262             }
263             v = (v << 3) | x
264         }
265         s = s[2:]
266         if v > 255 {
267             err = ErrSyntax
268             return
269         }
270         value = v
271     case '\\\\':
272         value = '\\\\'
273     case '\\', '\"':
274         if c != quote {
275             err = ErrSyntax
276             return
277         }
278         value = rune(c)
279     default:
280         err = ErrSyntax
281         return
282     }
283     tail = s
284     return
285 }
286
287 // Unquote interprets s as a single-quoted, double-quoted,
288 // or backquoted Go string literal, returning the string val
289 // that s quotes. (If s is single-quoted, it would be a Go
290 // character literal; Unquote returns the corresponding
291 // one-character string.)

```

```

292 func Unquote(s string) (t string, err error) {
293     n := len(s)
294     if n < 2 {
295         return "", ErrSyntax
296     }
297     quote := s[0]
298     if quote != s[n-1] {
299         return "", ErrSyntax
300     }
301     s = s[1 : n-1]
302
303     if quote == '`' {
304         if contains(s, '`') {
305             return "", ErrSyntax
306         }
307         return s, nil
308     }
309     if quote != '"' && quote != '\'' {
310         return "", ErrSyntax
311     }
312     if contains(s, '\n') {
313         return "", ErrSyntax
314     }
315
316     // Is it trivial? Avoid allocation.
317     if !contains(s, '\\') && !contains(s, quote) {
318         switch quote {
319             case '"':
320                 return s, nil
321             case '\'':
322                 r, size := utf8.DecodeRuneInString(s)
323                 if size == len(s) && (r != utf8.RuneError) {
324                     return s, nil
325                 }
326         }
327     }
328
329     var runeTmp [utf8.UTFMax]byte
330     buf := make([]byte, 0, 3*len(s)/2) // Try to avoid m
331     for len(s) > 0 {
332         c, multibyte, ss, err := UnquoteChar(s, quote)
333         if err != nil {
334             return "", err
335         }
336         s = ss
337         if c < utf8.RuneSelf || !multibyte {
338             buf = append(buf, byte(c))
339         } else {
340             n := utf8.EncodeRune(runeTmp[:], c)

```

```

341             buf = append(buf, runeTmp[:n]...)
342         }
343         if quote == '\'' && len(s) != 0 {
344             // single-quoted must be single char
345             return "", ErrSyntax
346         }
347     }
348     return string(buf), nil
349 }
350
351 // contains reports whether the string contains the byte c.
352 func contains(s string, c byte) bool {
353     for i := 0; i < len(s); i++ {
354         if s[i] == c {
355             return true
356         }
357     }
358     return false
359 }
360
361 // bsearch16 returns the smallest i such that a[i] >= x.
362 // If there is no such i, bsearch16 returns len(a).
363 func bsearch16(a []uint16, x uint16) int {
364     i, j := 0, len(a)
365     for i < j {
366         h := i + (j-i)/2
367         if a[h] < x {
368             i = h + 1
369         } else {
370             j = h
371         }
372     }
373     return i
374 }
375
376 // bsearch32 returns the smallest i such that a[i] >= x.
377 // If there is no such i, bsearch32 returns len(a).
378 func bsearch32(a []uint32, x uint32) int {
379     i, j := 0, len(a)
380     for i < j {
381         h := i + (j-i)/2
382         if a[h] < x {
383             i = h + 1
384         } else {
385             j = h
386         }
387     }
388     return i
389 }
390

```

```

391 // TODO: IsPrint is a local implementation of unicode.IsPrint
392 // to give the same answer. It allows this package not to de
393 // and therefore not pull in all the Unicode tables. If the
394 // at tossing unused tables, we could get rid of this implem
395 // That would be nice.
396
397 // IsPrint reports whether the rune is defined as printable
398 // the same definition as unicode.IsPrint: letters, numbers,
399 // symbols and ASCII space.
400 func IsPrint(r rune) bool {
401     // Fast check for Latin-1
402     if r <= 0xFF {
403         if 0x20 <= r && r <= 0x7E {
404             // All the ASCII is printable from s
405             return true
406         }
407         if 0xA1 <= r && r <= 0xFF {
408             // Similarly for ÿ through ÿ...
409             return r != 0xAD // ...except for th
410         }
411         return false
412     }
413
414     // Same algorithm, either on uint16 or uint32 value.
415     // First, find first i such that isPrint[i] >= x.
416     // This is the index of either the start or end of a
417     // The start is even (isPrint[i&^1]) and the end is
418     // If we find x in a range, make sure x is not in is
419
420     if 0 <= r && r < 1<<16 {
421         rr, isPrint, isNotPrint := uint16(r), isPrint
422         i := bsearch16(isPrint, rr)
423         if i >= len(isPrint) || rr < isPrint[i&^1] |
424             return false
425     }
426     j := bsearch16(isNotPrint, rr)
427     return j >= len(isNotPrint) || isNotPrint[j]
428 }
429
430 rr, isPrint, isNotPrint := uint32(r), isPrint32, isN
431 i := bsearch32(isPrint, rr)
432 if i >= len(isPrint) || rr < isPrint[i&^1] || isPrin
433     return false
434 }
435 if r >= 0x20000 {
436     return true
437 }
438 r -= 0x10000
439 j := bsearch16(isNotPrint, uint16(r))

```

```
440         return j >= len(isNotPrint) || isNotPrint[j] != uint
441     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strings/replace.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package strings
6
7 import "io"
8
9 // A Replacer replaces a list of strings with replacements.
10 type Replacer struct {
11     r replacer
12 }
13
14 // replacer is the interface that a replacement algorithm ne
15 type replacer interface {
16     Replace(s string) string
17     WriteString(w io.Writer, s string) (n int, err error)
18 }
19
20 // byteBitmap represents bytes which are sought for replacer
21 // byteBitmap is 256 bits wide, with a bit set for each old
22 // replaced.
23 type byteBitmap [256 / 32]uint32
24
25 func (m *byteBitmap) set(b byte) {
26     m[b>>5] |= uint32(1 << (b & 31))
27 }
28
29 // NewReplacer returns a new Replacer from a list of old, ne
30 // Replacements are performed in order, without overlapping
31 func NewReplacer(oldnew ...string) *Replacer {
32     if len(oldnew)%2 == 1 {
33         panic("strings.NewReplacer: odd argument cou
34     }
35
36     // Possible implementations.
37     var (
38         bb byteReplacer
39         bs byteStringReplacer
40         gen genericReplacer
41     )
42
43     allOldBytes, allNewBytes := true, true
44     for len(oldnew) > 0 {
```

```

45         old, new := oldnew[0], oldnew[1]
46         oldnew = oldnew[2:]
47         if len(old) != 1 {
48             allOldBytes = false
49         }
50         if len(new) != 1 {
51             allNewBytes = false
52         }
53
54         // generic
55         gen.p = append(gen.p, pair{old, new})
56
57         // byte -> string
58         if allOldBytes {
59             bs.old.set(old[0])
60             bs.new[old[0]] = []byte(new)
61         }
62
63         // byte -> byte
64         if allOldBytes && allNewBytes {
65             bb.old.set(old[0])
66             bb.new[old[0]] = new[0]
67         }
68     }
69
70     if allOldBytes && allNewBytes {
71         return &Replacer{r: &bb}
72     }
73     if allOldBytes {
74         return &Replacer{r: &bs}
75     }
76     return &Replacer{r: &gen}
77 }
78
79 // Replace returns a copy of s with all replacements perform
80 func (r *Replacer) Replace(s string) string {
81     return r.r.Replace(s)
82 }
83
84 // WriteString writes s to w with all replacements performed
85 func (r *Replacer) WriteString(w io.Writer, s string) (n int) {
86     return r.r.WriteString(w, s)
87 }
88
89 // genericReplacer is the fully generic (and least optimized)
90 // It's used as a fallback when nothing faster can be used.
91 type genericReplacer struct {
92     p []pair
93 }
94

```

```

95 type pair struct{ old, new string }
96
97 type appendSliceWriter struct {
98     b []byte
99 }
100
101 func (w *appendSliceWriter) Write(p []byte) (int, error) {
102     w.b = append(w.b, p...)
103     return len(p), nil
104 }
105
106 func (r *genericReplacer) Replace(s string) string {
107     // TODO(bradfitz): optimized version
108     n, _ := r.WriteString(discard, s)
109     w := appendSliceWriter{make([]byte, 0, n)}
110     r.WriteString(&w, s)
111     return string(w.b)
112 }
113
114 func (r *genericReplacer) WriteString(w io.Writer, s string)
115     lastEmpty := false // the last replacement was of th
116 Input:
117     // TODO(bradfitz): optimized version
118     for i := 0; i < len(s); {
119         for _, p := range r.p {
120             if p.old == "" && lastEmpty {
121                 // Don't let old match twice
122                 // (it doesn't advance the i
123                 // would otherwise loop fore
124                 continue
125             }
126             if HasPrefix(s[i:], p.old) {
127                 if p.new != "" {
128                     wn, err := w.Write([
129                     n += wn
130                     if err != nil {
131                         return n, er
132                     }
133                 }
134                 i += len(p.old)
135                 lastEmpty = p.old == ""
136                 continue Input
137             }
138         }
139         wn, err := w.Write([]byte{s[i]})
140         n += wn
141         if err != nil {
142             return n, err
143         }

```

```

144         i++
145     }
146
147     // Final empty match at end.
148     for _, p := range r.p {
149         if p.old == "" {
150             if p.new != "" {
151                 wn, err := w.Write([]byte(p.
152                     n += wn
153                     if err != nil {
154                         return n, err
155                     }
156                 }
157             }
158         }
159     }
160
161     return n, nil
162 }
163
164 // byteReplacer is the implementation that's used when all t
165 // and "new" values are single ASCII bytes.
166 type byteReplacer struct {
167     // old has a bit set for each old byte that should b
168     old byteBitmap
169
170     // replacement byte, indexed by old byte. only valid
171     // corresponding old bit is set.
172     new [256]byte
173 }
174
175 func (r *byteReplacer) Replace(s string) string {
176     var buf []byte // lazily allocated
177     for i := 0; i < len(s); i++ {
178         b := s[i]
179         if r.old[b>>5]&uint32(1<<(b&31)) != 0 {
180             if buf == nil {
181                 buf = []byte(s)
182             }
183             buf[i] = r.new[b]
184         }
185     }
186     if buf == nil {
187         return s
188     }
189     return string(buf)
190 }
191
192 func (r *byteReplacer) WriteString(w io.Writer, s string) (n

```

```

193     // TODO(bradfitz): use io.WriteString with slices of
194     bufsize := 32 << 10
195     if len(s) < bufsize {
196         bufsize = len(s)
197     }
198     buf := make([]byte, bufsize)
199
200     for len(s) > 0 {
201         ncopy := copy(buf, s[:])
202         s = s[ncopy:]
203         for i, b := range buf[:ncopy] {
204             if r.old[b>>5]&uint32(1<<(b&31)) !=
205                 buf[i] = r.new[b]
206         }
207     }
208     wn, err := w.Write(buf[:ncopy])
209     n += wn
210     if err != nil {
211         return n, err
212     }
213 }
214 return n, nil
215 }
216
217 // byteStringReplacer is the implementation that's used when
218 // "old" values are single ASCII bytes but the "new" values
219 // size.
220 type byteStringReplacer struct {
221     // old has a bit set for each old byte that should b
222     old byteBitmap
223
224     // replacement string, indexed by old byte. only val
225     // corresponding old bit is set.
226     new [256][]byte
227 }
228
229 func (r *byteStringReplacer) Replace(s string) string {
230     newSize := 0
231     anyChanges := false
232     for i := 0; i < len(s); i++ {
233         b := s[i]
234         if r.old[b>>5]&uint32(1<<(b&31)) != 0 {
235             anyChanges = true
236             newSize += len(r.new[b])
237         } else {
238             newSize++
239         }
240     }
241     if !anyChanges {
242         return s

```

```

243     }
244     buf := make([]byte, newSize)
245     bi := buf
246     for i := 0; i < len(s); i++ {
247         b := s[i]
248         if r.old[b>>5]&uint32(1<<(b&31)) != 0 {
249             n := copy(bi[:], r.new[b])
250             bi = bi[n:]
251         } else {
252             bi[0] = b
253             bi = bi[1:]
254         }
255     }
256     return string(buf)
257 }
258
259 // WriteString maintains one buffer that's at most 32KB. Th
260 // s are enumerated and the buffer is filled. If it reaches
261 // capacity or a byte has a replacement, the buffer is flush
262 func (r *byteStringReplacer) WriteString(w io.Writer, s stri
263 // TODO(bradfitz): use io.WriteString with slices of
264 bufsize := 32 << 10
265 if len(s) < bufsize {
266     bufsize = len(s)
267 }
268 buf := make([]byte, bufsize)
269 bi := buf[:0]
270
271 for i := 0; i < len(s); i++ {
272     b := s[i]
273     var new []byte
274     if r.old[b>>5]&uint32(1<<(b&31)) != 0 {
275         new = r.new[b]
276     } else {
277         bi = append(bi, b)
278     }
279     if len(bi) == cap(bi) || (len(bi) > 0 && len
280         nw, err := w.Write(bi)
281         n += nw
282         if err != nil {
283             return n, err
284         }
285         bi = buf[:0]
286     }
287     if len(new) > 0 {
288         nw, err := w.Write(new)
289         n += nw
290         if err != nil {
291             return n, err

```

```

292         }
293     }
294 }
295 if len(bi) > 0 {
296     nw, err := w.Write(bi)
297     n += nw
298     if err != nil {
299         return n, err
300     }
301 }
302 return n, nil
303 }
304
305 // strings is too low-level to import io/ioutil
306 var discard io.Writer = devNull(0)
307
308 type devNull int
309
310 func (devNull) Write(p []byte) (int, error) {
311     return len(p), nil
312 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/strings/strings.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package strings implements simple functions to manipulate
6 package strings
7
8 import (
9     "unicode"
10    "unicode/utf8"
11 )
12
13 // explode splits s into an array of UTF-8 sequences, one pe
14 // Invalid UTF-8 sequences become correct encodings of U+FFF
15 func explode(s string, n int) []string {
16     if n == 0 {
17         return nil
18     }
19     l := utf8.RuneCountInString(s)
20     if n <= 0 || n > l {
21         n = l
22     }
23     a := make([]string, n)
24     var size int
25     var ch rune
26     i, cur := 0, 0
27     for ; i+1 < n; i++ {
28         ch, size = utf8.DecodeRuneInString(s[cur:])
29         a[i] = string(ch)
30         cur += size
31     }
32     // add the rest, if there is any
33     if cur < len(s) {
34         a[i] = s[cur:]
35     }
36     return a
37 }
38
39 // Count counts the number of non-overlapping instances of s
40 func Count(s, sep string) int {
41     if sep == "" {
42         return utf8.RuneCountInString(s) + 1
43     }
44     c := sep[0]
```

```

45     l := len(sep)
46     n := 0
47     if l == 1 {
48         // special case worth making fast
49         for i := 0; i < len(s); i++ {
50             if s[i] == c {
51                 n++
52             }
53         }
54         return n
55     }
56     for i := 0; i+1 <= len(s); i++ {
57         if s[i] == c && s[i:i+1] == sep {
58             n++
59             i += l - 1
60         }
61     }
62     return n
63 }
64
65 // Contains returns true if substr is within s.
66 func Contains(s, substr string) bool {
67     return Index(s, substr) >= 0
68 }
69
70 // ContainsAny returns true if any Unicode code points in ch
71 func ContainsAny(s, chars string) bool {
72     return IndexAny(s, chars) >= 0
73 }
74
75 // ContainsRune returns true if the Unicode code point r is
76 func ContainsRune(s string, r rune) bool {
77     return IndexRune(s, r) >= 0
78 }
79
80 // Index returns the index of the first instance of sep in s
81 func Index(s, sep string) int {
82     n := len(sep)
83     if n == 0 {
84         return 0
85     }
86     c := sep[0]
87     if n == 1 {
88         // special case worth making fast
89         for i := 0; i < len(s); i++ {
90             if s[i] == c {
91                 return i
92             }
93         }
94     }
95     return -1

```

```

95     }
96     // n > 1
97     for i := 0; i+n <= len(s); i++ {
98         if s[i] == c && s[i:i+n] == sep {
99             return i
100        }
101    }
102    return -1
103 }
104
105 // LastIndex returns the index of the last instance of sep i
106 func LastIndex(s, sep string) int {
107     n := len(sep)
108     if n == 0 {
109         return len(s)
110     }
111     c := sep[0]
112     if n == 1 {
113         // special case worth making fast
114         for i := len(s) - 1; i >= 0; i-- {
115             if s[i] == c {
116                 return i
117             }
118         }
119         return -1
120     }
121     // n > 1
122     for i := len(s) - n; i >= 0; i-- {
123         if s[i] == c && s[i:i+n] == sep {
124             return i
125         }
126     }
127     return -1
128 }
129
130 // IndexRune returns the index of the first instance of the
131 // r, or -1 if rune is not present in s.
132 func IndexRune(s string, r rune) int {
133     switch {
134     case r < 0x80:
135         b := byte(r)
136         for i := 0; i < len(s); i++ {
137             if s[i] == b {
138                 return i
139             }
140         }
141     default:
142         for i, c := range s {
143             if c == r {

```

```

144             return i
145         }
146     }
147 }
148     return -1
149 }
150
151 // IndexAny returns the index of the first instance of any U
152 // from chars in s, or -1 if no Unicode code point from char
153 func IndexAny(s, chars string) int {
154     if len(chars) > 0 {
155         for i, c := range s {
156             for _, m := range chars {
157                 if c == m {
158                     return i
159                 }
160             }
161         }
162     }
163     return -1
164 }
165
166 // LastIndexAny returns the index of the last instance of an
167 // point from chars in s, or -1 if no Unicode code point fro
168 // present in s.
169 func LastIndexAny(s, chars string) int {
170     if len(chars) > 0 {
171         for i := len(s); i > 0; {
172             rune, size := utf8.DecodeLastRuneInS
173             i -= size
174             for _, m := range chars {
175                 if rune == m {
176                     return i
177                 }
178             }
179         }
180     }
181     return -1
182 }
183
184 // Generic split: splits after each instance of sep,
185 // including sepSave bytes of sep in the subarrays.
186 func genSplit(s, sep string, sepSave, n int) []string {
187     if n == 0 {
188         return nil
189     }
190     if sep == "" {
191         return explode(s, n)
192     }

```

```

193         if n < 0 {
194             n = Count(s, sep) + 1
195         }
196         c := sep[0]
197         start := 0
198         a := make([]string, n)
199         na := 0
200         for i := 0; i+len(sep) <= len(s) && na+1 < n; i++ {
201             if s[i] == c && (len(sep) == 1 || s[i:i+len(
202                 a[na] = s[start : i+sepSave]
203                 na++
204                 start = i + len(sep)
205                 i += len(sep) - 1
206             }
207         }
208         a[na] = s[start:]
209         return a[0 : na+1]
210     }
211
212     // SplitN slices s into substrings separated by sep and retu
213     // the substrings between those separators.
214     // If sep is empty, SplitN splits after each UTF-8 sequence.
215     // The count determines the number of substrings to return:
216     //   n > 0: at most n substrings; the last substring will be
217     //   n == 0: the result is nil (zero substrings)
218     //   n < 0: all substrings
219     func SplitN(s, sep string, n int) []string { return genSplit
220
221     // SplitAfterN slices s into substrings after each instance
222     // returns a slice of those substrings.
223     // If sep is empty, SplitAfterN splits after each UTF-8 sequ
224     // The count determines the number of substrings to return:
225     //   n > 0: at most n substrings; the last substring will be
226     //   n == 0: the result is nil (zero substrings)
227     //   n < 0: all substrings
228     func SplitAfterN(s, sep string, n int) []string {
229         return genSplit(s, sep, len(sep), n)
230     }
231
232     // Split slices s into all substrings separated by sep and r
233     // the substrings between those separators.
234     // If sep is empty, Split splits after each UTF-8 sequence.
235     // It is equivalent to SplitN with a count of -1.
236     func Split(s, sep string) []string { return genSplit(s, sep,
237
238     // SplitAfter slices s into all substrings after each instan
239     // returns a slice of those substrings.
240     // If sep is empty, SplitAfter splits after each UTF-8 seque
241     // It is equivalent to SplitAfterN with a count of -1.
242     func SplitAfter(s, sep string) []string {

```

```

243         return genSplit(s, sep, len(sep), -1)
244     }
245
246 // Fields splits the string s around each instance of one or
247 // characters, returning an array of substrings of s or an e
248 func Fields(s string) []string {
249     return FieldsFunc(s, unicode.IsSpace)
250 }
251
252 // FieldsFunc splits the string s at each run of Unicode cod
253 // and returns an array of slices of s. If all code points i
254 // string is empty, an empty slice is returned.
255 func FieldsFunc(s string, f func(rune) bool) []string {
256     // First count the fields.
257     n := 0
258     inField := false
259     for _, rune := range s {
260         wasInField := inField
261         inField = !f(rune)
262         if inField && !wasInField {
263             n++
264         }
265     }
266
267     // Now create them.
268     a := make([]string, n)
269     na := 0
270     fieldStart := -1 // Set to -1 when looking for start
271     for i, rune := range s {
272         if f(rune) {
273             if fieldStart >= 0 {
274                 a[na] = s[fieldStart:i]
275                 na++
276                 fieldStart = -1
277             }
278             } else if fieldStart == -1 {
279                 fieldStart = i
280             }
281     }
282     if fieldStart >= 0 { // Last field might end at EOF.
283         a[na] = s[fieldStart:]
284     }
285     return a
286 }
287
288 // Join concatenates the elements of a to create a single st
289 // sep is placed between elements in the resulting string.
290 func Join(a []string, sep string) string {
291     if len(a) == 0 {

```

```

292         return ""
293     }
294     if len(a) == 1 {
295         return a[0]
296     }
297     n := len(sep) * (len(a) - 1)
298     for i := 0; i < len(a); i++ {
299         n += len(a[i])
300     }
301
302     b := make([]byte, n)
303     bp := copy(b, a[0])
304     for _, s := range a[1:] {
305         bp += copy(b[bp:], sep)
306         bp += copy(b[bp:], s)
307     }
308     return string(b)
309 }
310
311 // HasPrefix tests whether the string s begins with prefix.
312 func HasPrefix(s, prefix string) bool {
313     return len(s) >= len(prefix) && s[0:len(prefix)] ==
314 }
315
316 // HasSuffix tests whether the string s ends with suffix.
317 func HasSuffix(s, suffix string) bool {
318     return len(s) >= len(suffix) && s[len(s)-len(suffix)
319 }
320
321 // Map returns a copy of the string s with all its character
322 // according to the mapping function. If mapping returns a n
323 // dropped from the string with no replacement.
324 func Map(mapping func(rune) rune, s string) string {
325     // In the worst case, the string can grow when mapp
326     // things unpleasant. But it's so rare we barge in
327     // fine. It could also shrink but that falls out na
328     maxbytes := len(s) // length of b
329     nbytes := 0        // number of bytes encoded in b
330     // The output buffer b is initialized on demand, the
331     // time a character differs.
332     var b []byte
333
334     for i, c := range s {
335         r := mapping(c)
336         if b == nil {
337             if r == c {
338                 continue
339             }
340             b = make([]byte, maxbytes)

```

```

341         nbytes = copy(b, s[:i])
342     }
343     if r >= 0 {
344         wid := 1
345         if r >= utf8.RuneSelf {
346             wid = utf8.RuneLen(r)
347         }
348         if nbytes+wid > maxbytes {
349             // Grow the buffer.
350             maxbytes = maxbytes*2 + utf8
351             nb := make([]byte, maxbytes)
352             copy(nb, b[0:nbytes])
353             b = nb
354         }
355         nbytes += utf8.EncodeRune(b[nbytes:r]
356     }
357 }
358 if b == nil {
359     return s
360 }
361 return string(b[0:nbytes])
362 }
363
364 // Repeat returns a new string consisting of count copies of
365 func Repeat(s string, count int) string {
366     b := make([]byte, len(s)*count)
367     bp := 0
368     for i := 0; i < count; i++ {
369         for j := 0; j < len(s); j++ {
370             b[bp] = s[j]
371             bp++
372         }
373     }
374     return string(b)
375 }
376
377 // ToUpper returns a copy of the string s with all Unicode 1
378 func ToUpper(s string) string { return Map(unicode.ToUpper,
379
380 // ToLower returns a copy of the string s with all Unicode 1
381 func ToLower(s string) string { return Map(unicode.ToLower,
382
383 // ToTitle returns a copy of the string s with all Unicode 1
384 func ToTitle(s string) string { return Map(unicode.ToTitle,
385
386 // ToUpperSpecial returns a copy of the string s with all Un
387 // upper case, giving priority to the special casing rules.
388 func ToUpperSpecial(_case unicode.SpecialCase, s string) str
389     return Map(func(r rune) rune { return _case.ToUpper(
390 }

```

```

391
392 // ToLowerSpecial returns a copy of the string s with all Un
393 // lower case, giving priority to the special casing rules.
394 func ToLowerSpecial(_case unicode.SpecialCase, s string) str
395     return Map(func(r rune) rune { return _case.ToLower(
396 }
397
398 // ToTitleSpecial returns a copy of the string s with all Un
399 // title case, giving priority to the special casing rules.
400 func ToTitleSpecial(_case unicode.SpecialCase, s string) str
401     return Map(func(r rune) rune { return _case.ToTitle(
402 }
403
404 // isSeparator reports whether the rune could mark a word bo
405 // TODO: update when package unicode captures more of the pr
406 func isSeparator(r rune) bool {
407     // ASCII alphanumerics and underscore are not separa
408     if r <= 0x7F {
409         switch {
410             case '0' <= r && r <= '9':
411                 return false
412             case 'a' <= r && r <= 'z':
413                 return false
414             case 'A' <= r && r <= 'Z':
415                 return false
416             case r == '_':
417                 return false
418         }
419         return true
420     }
421     // Letters and digits are not separators
422     if unicode.IsLetter(r) || unicode.IsDigit(r) {
423         return false
424     }
425     // Otherwise, all we can do for now is treat spaces
426     return unicode.IsSpace(r)
427 }
428
429 // BUG(r): The rule Title uses for word boundaries does not
430
431 // Title returns a copy of the string s with all Unicode let
432 // mapped to their title case.
433 func Title(s string) string {
434     // Use a closure here to remember state.
435     // Hackish but effective. Depends on Map scanning in
436     // the closure once per rune.
437     prev := ' '
438     return Map(
439         func(r rune) rune {

```

```

440             if isSeparator(prev) {
441                 prev = r
442                 return unicode.ToTitle(r)
443             }
444             prev = r
445             return r
446         },
447         s)
448     }
449
450     // TrimLeftFunc returns a slice of the string s with all lea
451     // Unicode code points c satisfying f(c) removed.
452     func TrimLeftFunc(s string, f func(rune) bool) string {
453         i := indexFunc(s, f, false)
454         if i == -1 {
455             return ""
456         }
457         return s[i:]
458     }
459
460     // TrimRightFunc returns a slice of the string s with all tr
461     // Unicode code points c satisfying f(c) removed.
462     func TrimRightFunc(s string, f func(rune) bool) string {
463         i := lastIndexFunc(s, f, false)
464         if i >= 0 && s[i] >= utf8.RuneSelf {
465             _, wid := utf8.DecodeRuneInString(s[i:])
466             i += wid
467         } else {
468             i++
469         }
470         return s[0:i]
471     }
472
473     // TrimFunc returns a slice of the string s with all leading
474     // and trailing Unicode code points c satisfying f(c) remove
475     func TrimFunc(s string, f func(rune) bool) string {
476         return TrimRightFunc(TrimLeftFunc(s, f), f)
477     }
478
479     // IndexFunc returns the index into s of the first Unicode
480     // code point satisfying f(c), or -1 if none do.
481     func IndexFunc(s string, f func(rune) bool) int {
482         return indexFunc(s, f, true)
483     }
484
485     // LastIndexFunc returns the index into s of the last
486     // Unicode code point satisfying f(c), or -1 if none do.
487     func LastIndexFunc(s string, f func(rune) bool) int {
488         return lastIndexFunc(s, f, true)

```

```

489 }
490
491 // indexFunc is the same as IndexFunc except that if
492 // truth==false, the sense of the predicate function is
493 // inverted.
494 func indexFunc(s string, f func(rune) bool, truth bool) int
495     start := 0
496     for start < len(s) {
497         wid := 1
498         r := rune(s[start])
499         if r >= utf8.RuneSelf {
500             r, wid = utf8.DecodeRuneInString(s[s
501         ]
502         if f(r) == truth {
503             return start
504         }
505         start += wid
506     }
507     return -1
508 }
509
510 // lastIndexFunc is the same as LastIndexFunc except that if
511 // truth==false, the sense of the predicate function is
512 // inverted.
513 func lastIndexFunc(s string, f func(rune) bool, truth bool)
514     for i := len(s); i > 0; {
515         r, size := utf8.DecodeLastRuneInString(s[0:i
516         i -= size
517         if f(r) == truth {
518             return i
519         }
520     }
521     return -1
522 }
523
524 func makeCutsetFunc(cutset string) func(rune) bool {
525     return func(r rune) bool { return IndexRune(cutset,
526 }
527
528 // Trim returns a slice of the string s with all leading and
529 // trailing Unicode code points contained in cutset removed.
530 func Trim(s string, cutset string) string {
531     if s == "" || cutset == "" {
532         return s
533     }
534     return TrimFunc(s, makeCutsetFunc(cutset))
535 }
536
537 // TrimLeft returns a slice of the string s with all leading
538 // Unicode code points contained in cutset removed.

```

```

539 func TrimLeft(s string, cutset string) string {
540     if s == "" || cutset == "" {
541         return s
542     }
543     return TrimLeftFunc(s, makeCutsetFunc(cutset))
544 }
545
546 // TrimRight returns a slice of the string s, with all trail
547 // Unicode code points contained in cutset removed.
548 func TrimRight(s string, cutset string) string {
549     if s == "" || cutset == "" {
550         return s
551     }
552     return TrimRightFunc(s, makeCutsetFunc(cutset))
553 }
554
555 // TrimSpace returns a slice of the string s, with all leadi
556 // and trailing white space removed, as defined by Unicode.
557 func TrimSpace(s string) string {
558     return TrimFunc(s, unicode.IsSpace)
559 }
560
561 // Replace returns a copy of the string s with the first n
562 // non-overlapping instances of old replaced by new.
563 // If n < 0, there is no limit on the number of replacements
564 func Replace(s, old, new string, n int) string {
565     if old == new || n == 0 {
566         return s // avoid allocation
567     }
568
569     // Compute number of replacements.
570     if m := Count(s, old); m == 0 {
571         return s // avoid allocation
572     } else if n < 0 || m < n {
573         n = m
574     }
575
576     // Apply replacements to buffer.
577     t := make([]byte, len(s)+n*(len(new)-len(old)))
578     w := 0
579     start := 0
580     for i := 0; i < n; i++ {
581         j := start
582         if len(old) == 0 {
583             if i > 0 {
584                 _, wid := utf8.DecodeRuneInS
585                 j += wid
586             }
587         } else {

```

```

588             j += Index(s[start:], old)
589         }
590         w += copy(t[w:], s[start:j])
591         w += copy(t[w:], new)
592         start = j + len(old)
593     }
594     w += copy(t[w:], s[start:])
595     return string(t[0:w])
596 }
597
598 // EqualFold reports whether s and t, interpreted as UTF-8 s
599 // are equal under Unicode case-folding.
600 func EqualFold(s, t string) bool {
601     for s != "" && t != "" {
602         // Extract first rune from each string.
603         var sr, tr rune
604         if s[0] < utf8.RuneSelf {
605             sr, s = rune(s[0]), s[1:]
606         } else {
607             r, size := utf8.DecodeRuneInString(s)
608             sr, s = r, s[size:]
609         }
610         if t[0] < utf8.RuneSelf {
611             tr, t = rune(t[0]), t[1:]
612         } else {
613             r, size := utf8.DecodeRuneInString(t)
614             tr, t = r, t[size:]
615         }
616
617         // If they match, keep going; if not, return
618
619         // Easy case.
620         if tr == sr {
621             continue
622         }
623
624         // Make sr < tr to simplify what follows.
625         if tr < sr {
626             tr, sr = sr, tr
627         }
628         // Fast check for ASCII.
629         if tr < utf8.RuneSelf && 'A' <= sr && sr <=
630             // ASCII, and sr is upper case. tr
631             if tr == sr+'a'-'A' {
632                 continue
633             }
634         return false
635     }
636 }

```

```
637             // General case. SimpleFold(x) returns the
638             // or wraps around to smaller values.
639             r := unicode.SimpleFold(sr)
640             for r != sr && r < tr {
641                 r = unicode.SimpleFold(r)
642             }
643             if r == tr {
644                 continue
645             }
646             return false
647         }
648
649         // One string is empty. Are both?
650         return s == t
651     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/sync/cond.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package sync
6
7 // Cond implements a condition variable, a rendezvous point
8 // for goroutines waiting for or announcing the occurrence
9 // of an event.
10 //
11 // Each Cond has an associated Locker L (often a *Mutex or *
12 // which must be held when changing the condition and
13 // when calling the Wait method.
14 type Cond struct {
15     L Locker // held while observing or changing the con
16     m Mutex // held to avoid internal races
17
18     // We must be careful to make sure that when Signal
19     // releases a semaphore, the corresponding acquire i
20     // executed by a goroutine that was already waiting
21     // the time of the call to Signal, not one that arri
22     // To ensure this, we segment waiting goroutines int
23     // generations punctuated by calls to Signal. Each
24     // Signal begins another generation if there are no
25     // left in older generations for it to wake. Becaus
26     // optimization (only begin another generation if th
27     // are no older goroutines left), we only need to ke
28     // of the two most recent generations, which we call
29     // and new.
30     oldWaiters int // number of waiters in old gener
31     oldSema    *uint32 // ... waiting on this semaphore
32
33     newWaiters int // number of waiters in new gener
34     newSema    *uint32 // ... waiting on this semaphore
35 }
36
37 // NewCond returns a new Cond with Locker l.
38 func NewCond(l Locker) *Cond {
39     return &Cond{L: l}
40 }
41
42 // Wait atomically unlocks c.L and suspends execution
43 // of the calling goroutine. After later resuming execution
44 // Wait locks c.L before returning. Unlike in other systems
```

```

45 // Wait cannot return unless awoken by Broadcast or Signal.
46 //
47 // Because c.L is not locked when Wait first resumes, the ca
48 // typically cannot assume that the condition is true when
49 // Wait returns.  Instead, the caller should Wait in a loop:
50 //
51 //     c.L.Lock()
52 //     for !condition() {
53 //         c.Wait()
54 //     }
55 //     ... make use of condition ...
56 //     c.L.Unlock()
57 //
58 func (c *Cond) Wait() {
59     c.m.Lock()
60     if c.newSema == nil {
61         c.newSema = new(uint32)
62     }
63     s := c.newSema
64     c.newWaiters++
65     c.m.Unlock()
66     c.L.Unlock()
67     runtime_Semacquire(s)
68     c.L.Lock()
69 }
70
71 // Signal wakes one goroutine waiting on c, if there is any.
72 //
73 // It is allowed but not required for the caller to hold c.L
74 // during the call.
75 func (c *Cond) Signal() {
76     c.m.Lock()
77     if c.oldWaiters == 0 && c.newWaiters > 0 {
78         // Retire old generation; rename new to old.
79         c.oldWaiters = c.newWaiters
80         c.oldSema = c.newSema
81         c.newWaiters = 0
82         c.newSema = nil
83     }
84     if c.oldWaiters > 0 {
85         c.oldWaiters--
86         runtime_Semrelease(c.oldSema)
87     }
88     c.m.Unlock()
89 }
90
91 // Broadcast wakes all goroutines waiting on c.
92 //
93 // It is allowed but not required for the caller to hold c.L
94 // during the call.

```

```
95 func (c *Cond) Broadcast() {
96     c.m.Lock()
97     // Wake both generations.
98     if c.oldWaiters > 0 {
99         for i := 0; i < c.oldWaiters; i++ {
100             runtime_Semrelease(c.oldSema)
101         }
102         c.oldWaiters = 0
103     }
104     if c.newWaiters > 0 {
105         for i := 0; i < c.newWaiters; i++ {
106             runtime_Semrelease(c.newSema)
107         }
108         c.newWaiters = 0
109         c.newSema = nil
110     }
111     c.m.Unlock()
112 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/sync/mutex.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package sync provides basic synchronization primitives such
6 // exclusion locks. Other than the Once and WaitGroup types
7 // for use by low-level library routines. Higher-level sync
8 // better done via channels and communication.
9 //
10 // Values containing the types defined in this package should
11 package sync
12
13 import "sync/atomic"
14
15 // A Mutex is a mutual exclusion lock.
16 // Mutexes can be created as part of other structures;
17 // the zero value for a Mutex is an unlocked mutex.
18 type Mutex struct {
19     state int32
20     sema  uint32
21 }
22
23 // A Locker represents an object that can be locked and unlocked.
24 type Locker interface {
25     Lock()
26     Unlock()
27 }
28
29 const (
30     mutexLocked = 1 << iota // mutex is locked
31     mutexWoken
32     mutexWaiterShift = iota
33 )
34
35 // Lock locks m.
36 // If the lock is already in use, the calling goroutine
37 // blocks until the mutex is available.
38 func (m *Mutex) Lock() {
39     // Fast path: grab unlocked mutex.
40     if atomic.CompareAndSwapInt32(&m.state, 0, mutexLocked) {
41         return
42     }
43
44     awake := false
```

```

45     for {
46         old := m.state
47         new := old | mutexLocked
48         if old&mutexLocked != 0 {
49             new = old + 1<<mutexWaiterShift
50         }
51         if awoke {
52             // The goroutine has been woken from
53             // so we need to reset the flag in e
54             new &^= mutexWoken
55         }
56         if atomic.CompareAndSwapInt32(&m.state, old,
57             if old&mutexLocked == 0 {
58                 break
59             }
60             runtime_Semacquire(&m.sema)
61             awoke = true
62         }
63     }
64 }
65
66 // Unlock unlocks m.
67 // It is a run-time error if m is not locked on entry to Unl
68 //
69 // A locked Mutex is not associated with a particular gorout
70 // It is allowed for one goroutine to lock a Mutex and then
71 // arrange for another goroutine to unlock it.
72 func (m *Mutex) Unlock() {
73     // Fast path: drop lock bit.
74     new := atomic.AddInt32(&m.state, -mutexLocked)
75     if (new+mutexLocked)&mutexLocked == 0 {
76         panic("sync: unlock of unlocked mutex")
77     }
78
79     old := new
80     for {
81         // If there are no waiters or a goroutine ha
82         // been woken or grabbed the lock, no need t
83         if old>>mutexWaiterShift == 0 || old&(mutexL
84             return
85         }
86         // Grab the right to wake someone.
87         new = (old - 1<<mutexWaiterShift) | mutexwok
88         if atomic.CompareAndSwapInt32(&m.state, old,
89             runtime_Semrelease(&m.sema)
90             return
91         }
92         old = m.state
93     }
94 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/sync/once.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package sync
6
7 import (
8     "sync/atomic"
9 )
10
11 // Once is an object that will perform exactly one action.
12 type Once struct {
13     m      Mutex
14     done uint32
15 }
16
17 // Do calls the function f if and only if the method is being
18 // first time with this receiver. In other words, given
19 //     var once Once
20 // if once.Do(f) is called multiple times, only the first call
21 // even if f has a different value in each invocation. A new
22 // Once is required for each function to execute.
23 //
24 // Do is intended for initialization that must be run exactly
25 // once. If the receiver is niladic, it may be necessary to use a function literal
26 // as arguments to a function to be invoked by Do:
27 //     config.once.Do(func() { config.init(filename) })
28 //
29 // Because no call to Do returns until the one call to f returns,
30 // Do to be called, it will deadlock.
31 //
32 func (o *Once) Do(f func()) {
33     if atomic.LoadUint32(&o.done) == 1 {
34         return
35     }
36     // Slow-path.
37     o.m.Lock()
38     defer o.m.Unlock()
39     if o.done == 0 {
40         f()
41         atomic.CompareAndSwapUint32(&o.done, 0, 1)
42     }
43 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/sync/runtime.go

```
1 // Copyright 2012 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package sync
6
7 // defined in package runtime
8
9 // Semacquire waits until *s > 0 and then atomically decreme
10 // It is intended as a simple sleep primitive for use by the
11 // library and should not be used directly.
12 func runtime_Semacquire(s *uint32)
13
14 // Semrelease atomically increments *s and notifies a waitin
15 // if one is blocked in Semacquire.
16 // It is intended as a simple wakeup primitive for use by th
17 // library and should not be used directly.
18 func runtime_Semrelease(s *uint32)
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/sync/rwmutex.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package sync
6
7 import "sync/atomic"
8
9 // An RWMutex is a reader/writer mutual exclusion lock.
10 // The lock can be held by an arbitrary number of readers
11 // or a single writer.
12 // RWMutexes can be created as part of other
13 // structures; the zero value for a RWMutex is
14 // an unlocked mutex.
15 type RWMutex struct {
16     w          Mutex // held if there are pending writ
17     writerSem  uint32 // semaphore for writers to wait
18     readerSem  uint32 // semaphore for readers to wait
19     readerCount int32 // number of pending readers
20     readerWait int32 // number of departing readers
21 }
22
23 const rwmutexMaxReaders = 1 << 30
24
25 // RLock locks rw for reading.
26 func (rw *RWMutex) RLock() {
27     if atomic.AddInt32(&rw.readerCount, 1) < 0 {
28         // A writer is pending, wait for it.
29         runtime_Semacquire(&rw.readerSem)
30     }
31 }
32
33 // RUnlock undoes a single RLock call;
34 // it does not affect other simultaneous readers.
35 // It is a run-time error if rw is not locked for reading
36 // on entry to RUnlock.
37 func (rw *RWMutex) RUnlock() {
38     if atomic.AddInt32(&rw.readerCount, -1) < 0 {
39         // A writer is pending.
40         if atomic.AddInt32(&rw.readerWait, -1) == 0
41             // The last reader unblocks the writ
42             runtime_Semrelease(&rw.writerSem)
43     }
44 }
```

```

45 }
46
47 // Lock locks rw for writing.
48 // If the lock is already locked for reading or writing,
49 // Lock blocks until the lock is available.
50 // To ensure that the lock eventually becomes available,
51 // a blocked Lock call excludes new readers from acquiring
52 // the lock.
53 func (rw *RWMutex) Lock() {
54     // First, resolve competition with other writers.
55     rw.w.Lock()
56     // Announce to readers there is a pending writer.
57     r := atomic.AddInt32(&rw.readerCount, -rwmutexMaxRea
58     // Wait for active readers.
59     if r != 0 && atomic.AddInt32(&rw.readerWait, r) != 0
60         runtime_Semacquire(&rw.writerSem)
61     }
62 }
63
64 // Unlock unlocks rw for writing. It is a run-time error if
65 // not locked for writing on entry to Unlock.
66 //
67 // As with Mutexes, a locked RWMutex is not associated with
68 // goroutine. One goroutine may RLock (Lock) an RWMutex and
69 // arrange for another goroutine to RUnlock (Unlock) it.
70 func (rw *RWMutex) Unlock() {
71     // Announce to readers there is no active writer.
72     r := atomic.AddInt32(&rw.readerCount, rwmutexMaxRead
73     // Unblock blocked readers, if any.
74     for i := 0; i < int(r); i++ {
75         runtime_Semrelease(&rw.readerSem)
76     }
77     // Allow other writers to proceed.
78     rw.w.Unlock()
79 }
80
81 // RLocker returns a Locker interface that implements
82 // the Lock and Unlock methods by calling rw.RLock and rw.RU
83 func (rw *RWMutex) RLocker() Locker {
84     return (*rlocker)(rw)
85 }
86
87 type rlocker RWMutex
88
89 func (r *rlocker) Lock() { (*RWMutex)(r).RLock() }
90 func (r *rlocker) Unlock() { (*RWMutex)(r).RUnlock() }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/sync/waitgroup.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package sync
6
7 import "sync/atomic"
8
9 // A WaitGroup waits for a collection of goroutines to finish
10 // The main goroutine calls Add to set the number of
11 // goroutines to wait for. Then each of the goroutines
12 // runs and calls Done when finished. At the same time,
13 // Wait can be used to block until all goroutines have finished
14 type WaitGroup struct {
15     m      Mutex
16     counter int32
17     waiters int32
18     sema   *uint32
19 }
20
21 // WaitGroup creates a new semaphore each time the old semaphore
22 // is released. This is to avoid the following race:
23 //
24 // G1: Add(1)
25 // G1: go G2()
26 // G1: Wait() // Context switch after Unlock() and before Semaphore
27 // G2: Done() // Release semaphore: sema == 1, waiters == 0.
28 // G3: Wait() // Finds counter == 0, waiters == 0, doesn't block
29 // G3: Add(1) // Makes counter == 1, waiters == 0.
30 // G3: go G4()
31 // G3: Wait() // G1 still hasn't run, G3 finds sema == 1, unblocks
32
33 // Add adds delta, which may be negative, to the WaitGroup counter
34 // If the counter becomes zero, all goroutines blocked on Wait
35 func (wg *WaitGroup) Add(delta int) {
36     v := atomic.AddInt32(&wg.counter, int32(delta))
37     if v < 0 {
38         panic("sync: negative WaitGroup count")
39     }
40     if v > 0 || atomic.LoadInt32(&wg.waiters) == 0 {
41         return
42     }
43     wg.m.Lock()
44     for i := int32(0); i < wg.waiters; i++ {
```

```

45             runtime_Semrelease(wg.sema)
46         }
47         wg.waiters = 0
48         wg.sema = nil
49         wg.m.Unlock()
50     }
51
52     // Done decrements the WaitGroup counter.
53     func (wg *WaitGroup) Done() {
54         wg.Add(-1)
55     }
56
57     // Wait blocks until the WaitGroup counter is zero.
58     func (wg *WaitGroup) Wait() {
59         if atomic.LoadInt32(&wg.counter) == 0 {
60             return
61         }
62         wg.m.Lock()
63         atomic.AddInt32(&wg.waiters, 1)
64         // This code is racing with the unlocked path in Add
65         // The code above modifies counter and then reads wa
66         // We must modify waiters and then read counter (the
67         // to avoid missing an Add.
68         if atomic.LoadInt32(&wg.counter) == 0 {
69             atomic.AddInt32(&wg.waiters, -1)
70             wg.m.Unlock()
71             return
72         }
73         if wg.sema == nil {
74             wg.sema = new(uint32)
75         }
76         s := wg.sema
77         wg.m.Unlock()
78         runtime_Semacquire(s)
79     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/sync/atomic/doc.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package atomic provides low-level atomic memory primitive
6 // useful for implementing synchronization algorithms.
7 //
8 // These functions require great care to be used correctly.
9 // Except for special, low-level applications, synchronizati
10 // done with channels or the facilities of the sync package.
11 // Share memory by communicating;
12 // don't communicate by sharing memory.
13 //
14 // The compare-and-swap operation, implemented by the Compar
15 // functions, is the atomic equivalent of:
16 //
17 //     if *val == old {
18 //         *val = new
19 //         return true
20 //     }
21 //     return false
22 //
23 package atomic
24
25 import (
26     "unsafe"
27 )
28
29 // BUG(rsc): On ARM, the 64-bit functions use instructions u
30 //
31 // On x86-32, the 64-bit functions use instructions unavaila
32
33 // CompareAndSwapInt32 executes the compare-and-swap operati
34 func CompareAndSwapInt32(val *int32, old, new int32) (swappe
35
36 // CompareAndSwapInt64 executes the compare-and-swap operati
37 func CompareAndSwapInt64(val *int64, old, new int64) (swappe
38
39 // CompareAndSwapUint32 executes the compare-and-swap operat
40 func CompareAndSwapUint32(val *uint32, old, new uint32) (swa
41
```

```
42 // CompareAndSwapUint64 executes the compare-and-swap operat
43 func CompareAndSwapUint64(val *uint64, old, new uint64) (swa
44
45 // CompareAndSwapUintptr executes the compare-and-swap opera
46 func CompareAndSwapUintptr(val *uintptr, old, new uintptr) (
47
48 // CompareAndSwapPointer executes the compare-and-swap opera
49 func CompareAndSwapPointer(val *unsafe.Pointer, old, new uns
50
51 // AddInt32 atomically adds delta to *val and returns the ne
52 func AddInt32(val *int32, delta int32) (new int32)
53
54 // AddUint32 atomically adds delta to *val and returns the n
55 func AddUint32(val *uint32, delta uint32) (new uint32)
56
57 // AddInt64 atomically adds delta to *val and returns the ne
58 func AddInt64(val *int64, delta int64) (new int64)
59
60 // AddUint64 atomically adds delta to *val and returns the n
61 func AddUint64(val *uint64, delta uint64) (new uint64)
62
63 // AddUintptr atomically adds delta to *val and returns the
64 func AddUintptr(val *uintptr, delta uintptr) (new uintptr)
65
66 // LoadInt32 atomically loads *addr.
67 func LoadInt32(addr *int32) (val int32)
68
69 // LoadInt64 atomically loads *addr.
70 func LoadInt64(addr *int64) (val int64)
71
72 // LoadUint32 atomically loads *addr.
73 func LoadUint32(addr *uint32) (val uint32)
74
75 // LoadUint64 atomically loads *addr.
76 func LoadUint64(addr *uint64) (val uint64)
77
78 // LoadUintptr atomically loads *addr.
79 func LoadUintptr(addr *uintptr) (val uintptr)
80
81 // LoadPointer atomically loads *addr.
82 func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
83
84 // StoreInt32 atomically stores val into *addr.
85 func StoreInt32(addr *int32, val int32)
86
87 // StoreInt64 atomically stores val into *addr.
88 func StoreInt64(addr *int64, val int64)
89
90 // StoreUint32 atomically stores val into *addr.
91 func StoreUint32(addr *uint32, val uint32)
```

```
92
93 // StoreUint64 atomically stores val into *addr.
94 func StoreUint64(addr *uint64, val uint64)
95
96 // StoreUintptr atomically stores val into *addr.
97 func StoreUintptr(addr *uintptr, val uintptr)
98
99 // StorePointer atomically stores val into *addr.
100 func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
101
102 // Helper for ARM. Linker will discard on other systems
103 func panic64() {
104     panic("sync/atomic: broken 64-bit atomic operations")
105 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/env_unix.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 // Unix environment variables.
8
9 package syscall
10
11 import "sync"
12
13 var (
14     // envOnce guards initialization by copyenv, which p
15     envOnce sync.Once
16
17     // envLock guards env and envs.
18     envLock sync.RWMutex
19
20     // env maps from an environment variable to its first
21     env map[string]int
22
23     // envs is provided by the runtime. elements are exp
24     // of the form "key=value".
25     envs []string
26 )
27
28 // setenv_c is provided by the runtime, but is a no-op if cg
29 // loaded.
30 func setenv_c(k, v string)
31
32 func copyenv() {
33     env = make(map[string]int)
34     for i, s := range envs {
35         for j := 0; j < len(s); j++ {
36             if s[j] == '=' {
37                 key := s[:j]
38                 if _, ok := env[key]; !ok {
39                     env[key] = i
40                 }
41                 break
42             }
43         }
44     }
45 }
```

```

42         }
43     }
44 }
45 }
46
47 func Getenv(key string) (value string, found bool) {
48     envOnce.Do(copyenv)
49     if len(key) == 0 {
50         return "", false
51     }
52
53     envLock.RLock()
54     defer envLock.RUnlock()
55
56     i, ok := env[key]
57     if !ok {
58         return "", false
59     }
60     s := envs[i]
61     for i := 0; i < len(s); i++ {
62         if s[i] == '=' {
63             return s[i+1:], true
64         }
65     }
66     return "", false
67 }
68
69 func Setenv(key, value string) error {
70     envOnce.Do(copyenv)
71     if len(key) == 0 {
72         return EINVAL
73     }
74
75     envLock.Lock()
76     defer envLock.Unlock()
77
78     i, ok := env[key]
79     kv := key + "=" + value
80     if ok {
81         envs[i] = kv
82     } else {
83         i = len(envs)
84         envs = append(envs, kv)
85     }
86     env[key] = i
87     setenv_c(key, value)
88     return nil
89 }
90
91 func Clearenv() {

```

```
92         envOnce.Do(copyenv) // prevent copyenv in Getenv/Set
93
94         envLock.Lock()
95         defer envLock.Unlock()
96
97         env = make(map[string]int)
98         envs = []string{}
99         // TODO(bradfitz): pass through to C
100    }
101
102    func Environ() []string {
103        envOnce.Do(copyenv)
104        envLock.RLock()
105        defer envLock.RUnlock()
106        a := make([]string, len(envs))
107        copy(a, envs)
108        return a
109    }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/exec_linux.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build linux
6
7 package syscall
8
9 import (
10     "unsafe"
11 )
12
13 type SysProcAttr struct {
14     Chroot      string    // Chroot.
15     Credential  *Credential // Credential.
16     Ptrace      bool      // Enable tracing.
17     Setsid      bool      // Create session.
18     Setpgid     bool      // Set process group ID to ne
19     Setctty     bool      // Set controlling terminal t
20     Noctty     bool      // Detach fd 0 from controllin
21     Pdeathsig   Signal    // Signal that the process wi
22 }
23
24 // Fork, dup fd onto 0..len(fd), and exec(argv0, argvv, envv
25 // If a dup or exec fails, write the errno error to pipe.
26 // (Pipe is close-on-exec so if exec succeeds, it will be cl
27 // In the child, this function must not acquire any locks, b
28 // they might have been locked at the time of the fork. Thi
29 // no rescheduling, no malloc calls, and no new stack segmen
30 // The calls to RawSyscall are okay because they are assembl
31 // functions that do not grow the stack.
32 func forkAndExecInChild(argv0 *byte, argv, envv []*byte, chr
33     // Declare all variables at top in case any
34     // declarations require heap allocation (e.g., err1)
35     var (
36         r1      uintptr
37         err1     Errno
38         nextfd  int
39         i       int
40     )
41
```

```

42 // guard against side effects of shuffling fds below
43 fd := make([]int, len(attr.Files))
44 for i, ufd := range attr.Files {
45     fd[i] = int(ufd)
46 }
47
48 // About to call fork.
49 // No more allocation or calls of non-assembly funct
50 r1, _, err1 = RawSyscall(SYS_FORK, 0, 0, 0)
51 if err1 != 0 {
52     return 0, err1
53 }
54
55 if r1 != 0 {
56     // parent; return PID
57     return int(r1), 0
58 }
59
60 // Fork succeeded, now in child.
61
62 // Parent death signal
63 if sys.Pdeathsig != 0 {
64     _, _, err1 = RawSyscall6(SYS_PRCTL, PR_SET_P
65     if err1 != 0 {
66         goto childerror
67     }
68
69     // Signal self if parent is already dead. Th
70     // duplicate signal in rare cases, but it wo
71     // using SIGKILL.
72     r1, _, _ = RawSyscall(SYS_GETPPID, 0, 0, 0)
73     if r1 == 1 {
74         pid, _, _ := RawSyscall(SYS_GETPID,
75         _, _, err1 := RawSyscall(SYS_KILL, p
76         if err1 != 0 {
77             goto childerror
78         }
79     }
80 }
81
82 // Enable tracing if requested.
83 if sys.Ptrace {
84     _, _, err1 = RawSyscall(SYS_PTRACE, uintptr(
85     if err1 != 0 {
86         goto childerror
87     }
88 }
89
90 // Session ID
91 if sys.Setsid {

```

```

92         _, _, err1 = RawSyscall(SYS_SETSID, 0, 0, 0)
93         if err1 != 0 {
94             goto childerror
95         }
96     }
97
98     // Set process group
99     if sys.Setpgid {
100         _, _, err1 = RawSyscall(SYS_SETPGID, 0, 0, 0)
101         if err1 != 0 {
102             goto childerror
103         }
104     }
105
106     // Chroot
107     if chroot != nil {
108         _, _, err1 = RawSyscall(SYS_CHROOT, uintptr(
109             if err1 != 0 {
110                 goto childerror
111             }
112         })
113     }
114
115     // User and groups
116     if cred := sys.Credential; cred != nil {
117         ngroups := uintptr(len(cred.Groups))
118         groups := uintptr(0)
119         if ngroups > 0 {
120             groups = uintptr(unsafe.Pointer(&cre
121         })
122         _, _, err1 = RawSyscall(SYS_SETGROUPS, ngrou
123         if err1 != 0 {
124             goto childerror
125         }
126         _, _, err1 = RawSyscall(SYS_SETGID, uintptr(
127         if err1 != 0 {
128             goto childerror
129         }
130         _, _, err1 = RawSyscall(SYS_SETUID, uintptr(
131         if err1 != 0 {
132             goto childerror
133         }
134     }
135
136     // Chdir
137     if dir != nil {
138         _, _, err1 = RawSyscall(SYS_CHDIR, uintptr(u
139         if err1 != 0 {
140             goto childerror

```

```

141     }
142
143     // Pass 1: look for fd[i] < i and move those up above
144     // so that pass 2 won't stomp on an fd it needs later
145     nextfd = int(len(fd))
146     if pipe < nextfd {
147         _, _, err1 = RawSyscall(SYS_DUP2, uintptr(pi
148         if err1 != 0 {
149             goto childerror
150         }
151         RawSyscall(SYS_FCNTL, uintptr(nextfd), F_SET
152         pipe = nextfd
153         nextfd++
154     }
155     for i = 0; i < len(fd); i++ {
156         if fd[i] >= 0 && fd[i] < int(i) {
157             _, _, err1 = RawSyscall(SYS_DUP2, ui
158             if err1 != 0 {
159                 goto childerror
160             }
161             RawSyscall(SYS_FCNTL, uintptr(nextfd
162             fd[i] = nextfd
163             nextfd++
164             if nextfd == pipe { // don't stomp o
165                 nextfd++
166             }
167         }
168     }
169
170     // Pass 2: dup fd[i] down onto i.
171     for i = 0; i < len(fd); i++ {
172         if fd[i] == -1 {
173             RawSyscall(SYS_CLOSE, uintptr(i), 0,
174             continue
175         }
176         if fd[i] == int(i) {
177             // dup2(i, i) won't clear close-on-e
178             // probably not elsewhere either.
179             _, _, err1 = RawSyscall(SYS_FCNTL, u
180             if err1 != 0 {
181                 goto childerror
182             }
183             continue
184         }
185         // The new fd is created NOT close-on-exec,
186         // which is exactly what we want.
187         _, _, err1 = RawSyscall(SYS_DUP2, uintptr(fd
188         if err1 != 0 {
189             goto childerror

```

```

190         }
191     }
192
193     // By convention, we don't close-on-exec the fds we
194     // started with, so if len(fd) < 3, close 0, 1, 2 as
195     // Programs that know they inherit fds >= 3 will need
196     // to set them close-on-exec.
197     for i = len(fd); i < 3; i++ {
198         RawSyscall(SYS_CLOSE, uintptr(i), 0, 0)
199     }
200
201     // Detach fd 0 from tty
202     if sys.Noctty {
203         _, _, err1 = RawSyscall(SYS_IOCTL, 0, uintptr
204         if err1 != 0 {
205             goto childerror
206         }
207     }
208
209     // Make fd 0 the tty
210     if sys.Setctty {
211         _, _, err1 = RawSyscall(SYS_IOCTL, 0, uintptr
212         if err1 != 0 {
213             goto childerror
214         }
215     }
216
217     // Time to exec.
218     _, _, err1 = RawSyscall(SYS_EXECVE,
219         uintptr(unsafe.Pointer(argv0)),
220         uintptr(unsafe.Pointer(&argv[0])),
221         uintptr(unsafe.Pointer(&envv[0])))
222
223 childerror:
224     // send error code on pipe
225     RawSyscall(SYS_WRITE, uintptr(pipe), uintptr(unsafe.
226     for {
227         RawSyscall(SYS_EXIT, 253, 0, 0)
228     }
229
230     // Calling panic is not actually safe,
231     // but the for loop above won't break
232     // and this shuts up the compiler.
233     panic("unreached")
234 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/exec_unix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 // Fork, exec, wait, etc.
8
9 package syscall
10
11 import (
12     "runtime"
13     "sync"
14     "unsafe"
15 )
16
17 // Lock synchronizing creation of new file descriptors with
18 //
19 // We want the child in a fork/exec sequence to inherit only
20 // file descriptors we intend. To do that, we mark all file
21 // descriptors close-on-exec and then, in the child, explici
22 // unmark the ones we want the exec'ed program to keep.
23 // Unix doesn't make this easy: there is, in general, no way
24 // allocate a new file descriptor close-on-exec. Instead yo
25 // have to allocate the descriptor and then mark it close-on
26 // If a fork happens between those two events, the child's e
27 // will inherit an unwanted file descriptor.
28 //
29 // This lock solves that race: the create new fd/mark close-
30 // operation is done holding ForkLock for reading, and the f
31 // is done holding ForkLock for writing. At least, that's t
32 // There are some complications.
33 //
34 // Some system calls that create new file descriptors can bl
35 // for arbitrarily long times: open on a hung NFS server or
36 // pipe, accept on a socket, and so on. We can't reasonably
37 // the lock across those operations.
38 //
39 // It is worse to inherit some file descriptors than others.
40 // If a non-malicious child accidentally inherits an open or
41 // that's not a big deal. On the other hand, if a long-live
```

```

42 // accidentally inherits the write end of a pipe, then the r
43 // of that pipe will not see EOF until that child exits, pot
44 // causing the parent program to hang. This is a common pro
45 // in threaded C programs that use popen.
46 //
47 // Luckily, the file descriptors that are most important not
48 // inherit are not the ones that can take an arbitrarily lon
49 // to create: pipe returns instantly, and the net package us
50 // non-blocking I/O to accept on a listening socket.
51 // The rules for which file descriptor-creating operations u
52 // ForkLock are as follows:
53 //
54 // 1) Pipe.      Does not block.  Use the ForkLock.
55 // 2) Socket.   Does not block.  Use the ForkLock.
56 // 3) Accept.   If using non-blocking mode, use the ForkLock.
57 //             Otherwise, live with the race.
58 // 4) Open.     Can block.  Use O_CLOEXEC if available (Linux
59 //             Otherwise, live with the race.
60 // 5) Dup.      Does not block.  Use the ForkLock.
61 //             On Linux, could use fcntl F_DUPFD_CLOEXEC
62 //             instead of the ForkLock, but only for dup(fd,
63
64 var ForkLock sync.RWMutex
65
66 // Convert array of string to array
67 // of NUL-terminated byte pointer.
68 func StringSlicePtr(ss []string) []*byte {
69     bb := make([]*byte, len(ss)+1)
70     for i := 0; i < len(ss); i++ {
71         bb[i] = StringBytePtr(ss[i])
72     }
73     bb[len(ss)] = nil
74     return bb
75 }
76
77 func CloseOnExec(fd int) { fcntl(fd, F_SETFD, FD_CLOEXEC) }
78
79 func SetNonblock(fd int, nonblocking bool) (err error) {
80     flag, err := fcntl(fd, F_GETFL, 0)
81     if err != nil {
82         return err
83     }
84     if nonblocking {
85         flag |= O_NONBLOCK
86     } else {
87         flag &= ^O_NONBLOCK
88     }
89     _, err = fcntl(fd, F_SETFL, flag)
90     return err
91 }

```

```

92
93 // Credential holds user and group identities to be assumed
94 // by a child process started by StartProcess.
95 type Credential struct {
96     Uid    uint32 // User ID.
97     Gid    uint32 // Group ID.
98     Groups []uint32 // Supplementary group IDs.
99 }
100
101 // ProcAttr holds attributes that will be applied to a new p
102 // by StartProcess.
103 type ProcAttr struct {
104     Dir    string // Current working directory.
105     Env    []string // Environment.
106     Files []uintptr // File descriptors.
107     Sys    *SysProcAttr
108 }
109
110 var zeroProcAttr ProcAttr
111 var zeroSysProcAttr SysProcAttr
112
113 func forkExec(argv0 string, argv []string, attr *ProcAttr) (
114     var p [2]int
115     var n int
116     var err1 Errno
117     var wstatus WaitStatus
118
119     if attr == nil {
120         attr = &zeroProcAttr
121     }
122     sys := attr.Sys
123     if sys == nil {
124         sys = &zeroSysProcAttr
125     }
126
127     p[0] = -1
128     p[1] = -1
129
130     // Convert args to C form.
131     argv0p := StringBytePtr(argv0)
132     argvp := StringSlicePtr(argv)
133     envvp := StringSlicePtr(attr.Env)
134
135     if runtime.GOOS == "freebsd" && len(argv[0]) > len(a
136         argvp[0] = argv0p
137     }
138
139     var chroot *byte
140     if sys.Chroot != "" {

```

```

141         chroot = StringBytePtr(sys.Chroot)
142     }
143     var dir *byte
144     if attr.Dir != "" {
145         dir = StringBytePtr(attr.Dir)
146     }
147
148     // Acquire the fork lock so that no other threads
149     // create new fds that are not yet close-on-exec
150     // before we fork.
151     ForkLock.Lock()
152
153     // Allocate child status pipe close on exec.
154     if err = Pipe(p[0:]); err != nil {
155         goto error
156     }
157     if _, err = fcntl(p[0], F_SETFD, FD_CLOEXEC); err !=
158         goto error
159     }
160     if _, err = fcntl(p[1], F_SETFD, FD_CLOEXEC); err !=
161         goto error
162     }
163
164     // Kick off child.
165     pid, err1 = forkAndExecInChild(argv0p, argvp, envvp,
166     if err1 != 0 {
167         err = Errno(err1)
168         goto error
169     }
170     ForkLock.Unlock()
171
172     // Read child error status from pipe.
173     Close(p[1])
174     n, err = read(p[0], (*byte)(unsafe.Pointer(&err1)),
175     Close(p[0])
176     if err != nil || n != 0 {
177         if n == int(unsafe.Sizeof(err1)) {
178             err = Errno(err1)
179         }
180         if err == nil {
181             err = EPIPE
182         }
183
184         // Child failed; wait for it to exit, to mak
185         // the zombies don't accumulate.
186         _, err1 := Wait4(pid, &wstatus, 0, nil)
187         for err1 == EINTR {
188             _, err1 = Wait4(pid, &wstatus, 0, ni
189     }

```

```

190             return 0, err
191         }
192
193         // Read got EOF, so pipe closed on exec, so exec suc
194         return pid, nil
195
196     error:
197         if p[0] >= 0 {
198             Close(p[0])
199             Close(p[1])
200         }
201         ForkLock.Unlock()
202         return 0, err
203     }
204
205     // Combination of fork and exec, careful to be thread safe.
206     func ForkExec(argv0 string, argv []string, attr *ProcAttr) (
207         return forkExec(argv0, argv, attr)
208     }
209
210     // StartProcess wraps ForkExec for package os.
211     func StartProcess(argv0 string, argv []string, attr *ProcAtt
212         pid, err = forkExec(argv0, argv, attr)
213         return pid, 0, err
214     }
215
216     // Ordinary exec.
217     func Exec(argv0 string, argv []string, envv []string) (err e
218         _, _, err1 := RawSyscall(SYS_EXECVE,
219             uintptr(unsafe.Pointer(StringBytePtr(argv0)))
220             uintptr(unsafe.Pointer(&StringSlicePtr(argv)
221             uintptr(unsafe.Pointer(&StringSlicePtr(envv)
222         return Errno(err1)
223     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/syscall/lst_linux.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Linux socket filter
6
7 package syscall
8
9 import (
10     "unsafe"
11 )
12
13 func LsfStmt(code, k int) *SockFilter {
14     return &SockFilter{Code: uint16(code), K: uint32(k)}
15 }
16
17 func LsfJump(code, k, jt, jf int) *SockFilter {
18     return &SockFilter{Code: uint16(code), Jt: uint8(jt)}
19 }
20
21 func LsfSocket(ifindex, proto int) (int, error) {
22     var lsall SockaddrLinklayer
23     s, e := Socket(AF_PACKET, SOCK_RAW, proto)
24     if e != nil {
25         return 0, e
26     }
27     p := (*[2]byte)(unsafe.Pointer(&lsall.Protocol))
28     p[0] = byte(proto >> 8)
29     p[1] = byte(proto)
30     lsall.Ifindex = ifindex
31     e = Bind(s, &lsall)
32     if e != nil {
33         Close(s)
34         return 0, e
35     }
36     return s, nil
37 }
38
39 type iflags struct {
40     name [IFNAMSIZ]byte
41     flags uint16
42 }
43
44 func SetLsfPromisc(name string, m bool) error {
```

```

45     s, e := Socket(AF_INET, SOCK_DGRAM, 0)
46     if e != nil {
47         return e
48     }
49     defer Close(s)
50     var ifl iflags
51     copy(ifl.name[:], []byte(name))
52     _, _, ep := Syscall(SYS_IOCTL, uintptr(s), SIOCGIFFL
53     if ep != 0 {
54         return Errno(ep)
55     }
56     if m {
57         ifl.flags |= uint16(IFF_PROMISC)
58     } else {
59         ifl.flags &= ^uint16(IFF_PROMISC)
60     }
61     _, _, ep = Syscall(SYS_IOCTL, uintptr(s), SIOCSIFFLA
62     if ep != 0 {
63         return Errno(ep)
64     }
65     return nil
66 }
67
68 func AttachLsf(fd int, i []SockFilter) error {
69     var p SockFprog
70     p.Len = uint16(len(i))
71     p.Filter = (*SockFilter)(unsafe.Pointer(&i[0]))
72     return setsockopt(fd, SOL_SOCKET, SO_ATTACH_FILTER,
73 }
74
75 func DetachLsf(fd int) error {
76     var dummy int
77     return setsockopt(fd, SOL_SOCKET, SO_DETACH_FILTER,
78 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/netlink_linux.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Netlink sockets and messages
6
7 package syscall
8
9 import (
10     "unsafe"
11 )
12
13 // Round the length of a netlink message up to align it prop
14 func nlmAlignOf(msglen int) int {
15     return (msglen + NLMSG_ALIGNTO - 1) & ^(NLMSG_ALIGNTO -
16 )
17
18 // Round the length of a netlink route attribute up to align
19 // properly.
20 func rtaAlignOf(attrlen int) int {
21     return (attrlen + RTA_ALIGNTO - 1) & ^(RTA_ALIGNTO -
22 )
23
24 // NetlinkRouteRequest represents the request message to rec
25 // routing and link states from the kernel.
26 type NetlinkRouteRequest struct {
27     Header NlMsgHdr
28     Data   RtGenmsg
29 }
30
31 func (rr *NetlinkRouteRequest) toWireFormat() []byte {
32     b := make([]byte, rr.Header.Len)
33     b[0] = byte(rr.Header.Len)
34     b[1] = byte(rr.Header.Len >> 8)
35     b[2] = byte(rr.Header.Len >> 16)
36     b[3] = byte(rr.Header.Len >> 24)
37     b[4] = byte(rr.Header.Type)
38     b[5] = byte(rr.Header.Type >> 8)
39     b[6] = byte(rr.Header.Flags)
40     b[7] = byte(rr.Header.Flags >> 8)
41     b[8] = byte(rr.Header.Seq)
```

```

42         b[9] = byte(rr.Header.Seq >> 8)
43         b[10] = byte(rr.Header.Seq >> 16)
44         b[11] = byte(rr.Header.Seq >> 24)
45         b[12] = byte(rr.Header.Pid)
46         b[13] = byte(rr.Header.Pid >> 8)
47         b[14] = byte(rr.Header.Pid >> 16)
48         b[15] = byte(rr.Header.Pid >> 24)
49         b[16] = byte(rr.Data.Family)
50         return b
51     }
52
53     func newNetlinkRouteRequest(proto, seq, family int) []byte {
54         rr := &NetlinkRouteRequest{}
55         rr.Header.Len = NLMSG_HDRLEN + SizeofRtGenmsg
56         rr.Header.Type = uint16(proto)
57         rr.Header.Flags = NLM_F_DUMP | NLM_F_REQUEST
58         rr.Header.Seq = uint32(seq)
59         rr.Data.Family = uint8(family)
60         return rr.toWireFormat()
61     }
62
63     // NetlinkRIB returns routing information base, as known as
64     // which consists of network facility information, states an
65     // parameters.
66     func NetlinkRIB(proto, family int) ([]byte, error) {
67         var (
68             lsanl SockaddrNetlink
69             tab []byte
70         )
71
72         s, e := Socket(AF_NETLINK, SOCK_RAW, 0)
73         if e != nil {
74             return nil, e
75         }
76         defer Close(s)
77
78         lsanl.Family = AF_NETLINK
79         e = Bind(s, &lsanl)
80         if e != nil {
81             return nil, e
82         }
83
84         seq := 1
85         wb := newNetlinkRouteRequest(proto, seq, family)
86         e = Sendto(s, wb, 0, &lsanl)
87         if e != nil {
88             return nil, e
89         }
90
91         for {

```

```

92         var (
93             rb []byte
94             nr int
95             lsa Sockaddr
96         )
97
98         rb = make([]byte, Getpagesize())
99         nr, _, e = Recvfrom(s, rb, 0)
100        if e != nil {
101            return nil, e
102        }
103        if nr < NLMSG_HDRLEN {
104            return nil, EINVAL
105        }
106        rb = rb[:nr]
107        tab = append(tab, rb...)
108
109        msgs, _ := ParseNetlinkMessage(rb)
110        for _, m := range msgs {
111            if lsa, e = Getsockname(s); e != nil
112                return nil, e
113            }
114            switch v := lsa.(type) {
115            case *SockaddrNetlink:
116                if m.Header.Seq != uint32(se
117                    return nil, EINVAL
118                }
119            default:
120                return nil, EINVAL
121            }
122            if m.Header.Type == NLMSG_DONE {
123                goto done
124            }
125            if m.Header.Type == NLMSG_ERROR {
126                return nil, EINVAL
127            }
128        }
129    }
130
131    done:
132        return tab, nil
133    }
134
135    // NetlinkMessage represents the netlink message.
136    type NetlinkMessage struct {
137        Header NlMsgHdr
138        Data []byte
139    }
140

```

```

141 // ParseNetlinkMessage parses buf as netlink messages and re
142 // the slice containing the NetlinkMessage structs.
143 func ParseNetlinkMessage(buf []byte) ([]NetlinkMessage, erro
144     var (
145         h    *NlMsgHdr
146         dbuf []byte
147         dlen int
148         e    error
149         msgs []NetlinkMessage
150     )
151
152     for len(buf) >= NLMSG_HDRLEN {
153         h, dbuf, dlen, e = netlinkMessageHeaderAndDa
154         if e != nil {
155             break
156         }
157         m := NetlinkMessage{}
158         m.Header = *h
159         m.Data = dbuf[:h.Len-NLMSG_HDRLEN]
160         msgs = append(msgs, m)
161         buf = buf[dlen:]
162     }
163
164     return msgs, e
165 }
166
167 func netlinkMessageHeaderAndData(buf []byte) (*NlMsgHdr, []b
168     h := (*NlMsgHdr)(unsafe.Pointer(&buf[0]))
169     if h.Len < NLMSG_HDRLEN || int(h.Len) > len(buf) {
170         return nil, nil, 0, EINVAL
171     }
172     return h, buf[NLMSG_HDRLEN:], nlmAlignOf(int(h.Len))
173 }
174
175 // NetlinkRouteAttr represents the netlink route attribute.
176 type NetlinkRouteAttr struct {
177     Attr RtAttr
178     Value []byte
179 }
180
181 // ParseNetlinkRouteAttr parses msg's payload as netlink rou
182 // attributes and returns the slice containing the NetlinkRo
183 // structs.
184 func ParseNetlinkRouteAttr(msg *NetlinkMessage) ([]NetlinkRo
185     var (
186         buf    []byte
187         a      *RtAttr
188         alen  int
189         vbuf   []byte

```

```

190         e    error
191         attrs []NetlinkRouteAttr
192     )
193
194     switch msg.Header.Type {
195     case RTM_NEWLINK, RTM_DELLINK:
196         buf = msg.Data[SizeofIfInfomsg:]
197     case RTM_NEWADDR, RTM_DELADDR:
198         buf = msg.Data[SizeofIfAddrmsg:]
199     case RTM_NEWROUTE, RTM_DELROUTE:
200         buf = msg.Data[SizeofRtMsg:]
201     default:
202         return nil, EINVAL
203     }
204
205     for len(buf) >= SizeofRtAttr {
206         a, vbuf, alen, e = netlinkRouteAttrAndValue(
207             if e != nil {
208                 break
209             }
210             ra := NetlinkRouteAttr{}
211             ra.Attr = *a
212             ra.Value = vbuf[:a.Len-SizeofRtAttr]
213             attrs = append(attrs, ra)
214             buf = buf[alen:]
215         }
216
217     return attrs, nil
218 }
219
220 func netlinkRouteAttrAndValue(buf []byte) (*RtAttr, []byte,
221     h := (*RtAttr)(unsafe.Pointer(&buf[0]))
222     if h.Len < SizeofRtAttr || int(h.Len) > len(buf) {
223         return nil, nil, 0, EINVAL
224     }
225     return h, buf[SizeofRtAttr:], rtaAlignOf(int(h.Len))
226 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/sockcmsg_linux.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Socket control messages
6
7 package syscall
8
9 import (
10     "unsafe"
11 )
12
13 // UnixCredentials encodes credentials into a socket control
14 // for sending to another process. This can be used for
15 // authentication.
16 func UnixCredentials(ucred *Ucred) []byte {
17     buf := make([]byte, CmsgSpace(SizeofUcred))
18     cmsg := (*CmsgHdr)(unsafe.Pointer(&buf[0]))
19     cmsg.Level = SOL_SOCKET
20     cmsg.Type = SCM_CREDENTIALS
21     cmsg.SetLen(CmsgLen(SizeofUcred))
22     *((*Ucred)(cmsgData(cmsg))) = *ucred
23     return buf
24 }
25
26 // ParseUnixCredentials decodes a socket control message tha
27 // credentials in a Ucred structure. To receive such a messa
28 // SO_PASSCRED option must be enabled on the socket.
29 func ParseUnixCredentials(msg *SocketControlMessage) (*Ucred
30     if msg.Header.Level != SOL_SOCKET {
31         return nil, EINVAL
32     }
33     if msg.Header.Type != SCM_CREDENTIALS {
34         return nil, EINVAL
35     }
36     ucred := (*Ucred)(unsafe.Pointer(&msg.Data[0]))
37     return &ucred, nil
38 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative

Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/sockcmsg_unix.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 // Socket control messages
8
9 package syscall
10
11 import (
12     "unsafe"
13 )
14
15 // Round the length of a raw sockaddr up to align it properly
16 func cmsgAlignOf(salen int) int {
17     salign := sizeofPtr
18     // NOTE: It seems like 64-bit Darwin kernel still re
19     // aligned access to BSD subsystem.
20     if darwinAMD64 {
21         salign = 4
22     }
23     if salen == 0 {
24         return salign
25     }
26     return (salen + salign - 1) & ^(salign - 1)
27 }
28
29 // CmsgLen returns the value to store in the Len field of th
30 // structure, taking into account any necessary alignment.
31 func CmsgLen(datalen int) int {
32     return cmsgAlignOf(SizeofCmsgHdr) + datalen
33 }
34
35 // CmsgSpace returns the number of bytes an ancillary elemen
36 // payload of the passed data length occupies.
37 func CmsgSpace(datalen int) int {
38     return cmsgAlignOf(SizeofCmsgHdr) + cmsgAlignOf(data
39 }
40
41 func cmsgData(cmsg *CmsgHdr) unsafe.Pointer {
```

```

42         return unsafe.Pointer(uintptr(unsafe.Pointer(cmsg)))
43     }
44
45     type SocketControlMessage struct {
46         Header Cmsghdr
47         Data []byte
48     }
49
50     func ParseSocketControlMessage(buf []byte) ([]SocketControlM
51         var (
52             h      *Cmsghdr
53             dbuf   []byte
54             e      error
55             cmsgs  []SocketControlMessage
56         )
57
58         for len(buf) >= CmsgLen(0) {
59             h, dbuf, e = socketControlMessageHeaderAndDa
60             if e != nil {
61                 break
62             }
63             m := SocketControlMessage{}
64             m.Header = *h
65             m.Data = dbuf[:int(h.Len)-cmsgAlignOf(Sizeof
66             cmsgs = append(cmsgs, m)
67             buf = buf[cmsgAlignOf(int(h.Len)):]
68         }
69
70         return cmsgs, e
71     }
72
73     func socketControlMessageHeaderAndData(buf []byte) (*Cmsghdr
74         h := (*Cmsghdr)(unsafe.Pointer(&buf[0]))
75         if h.Len < SizeofCmsghdr || int(h.Len) > len(buf) {
76             return nil, nil, EINVAL
77         }
78         return h, buf[cmsgAlignOf(SizeofCmsghdr):], nil
79     }
80
81     // UnixRights encodes a set of open file descriptors into a
82     // control message for sending to another process.
83     func UnixRights(fds ...int) []byte {
84         datalen := len(fds) * 4
85         buf := make([]byte, CmsgSpace(datalen))
86         cmsg := (*Cmsghdr)(unsafe.Pointer(&buf[0]))
87         cmsg.Level = SOL_SOCKET
88         cmsg.Type = SCM_RIGHTS
89         cmsg.SetLen(CmsgLen(datalen))
90
91         data := uintptr(cmsgData(cmsg))

```

```

92         for _, fd := range fds {
93             *(*int32)(unsafe.Pointer(data)) = int32(fd)
94             data += 4
95         }
96     }
97     return buf
98 }
99
100 // ParseUnixRights decodes a socket control message that con
101 // integer array of open file descriptors from another proce
102 func ParseUnixRights(msg *SocketControlMessage) ([]int, erro
103     if msg.Header.Level != SOL_SOCKET {
104         return nil, EINVAL
105     }
106     if msg.Header.Type != SCM_RIGHTS {
107         return nil, EINVAL
108     }
109     fds := make([]int, len(msg.Data)>>2)
110     for i, j := 0, 0; i < len(msg.Data); i += 4 {
111         fds[j] = int(*(*int32)(unsafe.Pointer(&msg.D
112             j++
113     }
114     return fds, nil
115 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/syscall/str.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package syscall
6
7 func itoa(val int) string { // do it here rather than with f
8     if val < 0 {
9         return "-" + itoa(-val)
10    }
11    var buf [32]byte // big enough for int64
12    i := len(buf) - 1
13    for val >= 10 {
14        buf[i] = byte(val%10 + '0')
15        i--
16        val /= 10
17    }
18    buf[i] = byte(val + '0')
19    return string(buf[i:])
20 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/syscall/syscall.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package syscall contains an interface to the low-level op
6 // primitives. The details vary depending on the underlying
7 // Its primary use is inside other packages that provide a n
8 // interface to the system, such as "os", "time" and "net".
9 // packages rather than this one if you can.
10 // For details of the functions and data types in this packa
11 // the manuals for the appropriate operating system.
12 // These calls return err == nil to indicate success; otherw
13 // err is an operating system error describing the failure.
14 // On most systems, that error has type syscall.Errno.
15 package syscall
16
17 // StringByteSlice returns a NUL-terminated slice of bytes
18 // containing the text of s.
19 func StringByteSlice(s string) []byte {
20     a := make([]byte, len(s)+1)
21     copy(a, s)
22     return a
23 }
24
25 // StringBytePtr returns a pointer to a NUL-terminated array
26 // containing the text of s.
27 func StringBytePtr(s string) *byte { return &StringByteSlice
28
29 // Single-word zero for use when we need a valid pointer to
30 // See mksyscall.pl.
31 var _zero uintptr
32
33 func (ts *Timespec) Unix() (sec int64, nsec int64) {
34     return int64(ts.Sec), int64(ts.Nsec)
35 }
36
37 func (tv *Timeval) Unix() (sec int64, nsec int64) {
38     return int64(tv.Sec), int64(tv.Usec) * 1000
39 }
40
41 func (ts *Timespec) Nano() int64 {
42     return int64(ts.Sec)*1e9 + int64(ts.Nsec)
43 }
44
```

```
45 func (tv *Timeval) Nano() int64 {  
46     return int64(tv.Sec)*1e9 + int64(tv.Usec)*1000  
47 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/syscall_linux.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Linux system calls.
6 // This file is compiled as ordinary Go code,
7 // but it is also input to mksyscall,
8 // which parses the //sys lines and generates system call st
9 // Note that sometimes we use a lowercase //sys name and
10 // wrap it in our own nicer implementation.
11
12 package syscall
13
14 import "unsafe"
15
16 /*
17  * Wrapped
18  */
19
20 //sys  open(path string, mode int, perm uint32) (fd int, er
21 func Open(path string, mode int, perm uint32) (fd int, err e
22     return open(path, mode|O_LARGEFILE, perm)
23 }
24
25 //sys  openat(dirfd int, path string, flags int, mode uint3
26 func Openat(dirfd int, path string, flags int, mode uint32)
27     return openat(dirfd, path, flags|O_LARGEFILE, mode)
28 }
29
30 //sysnb pipe(p *[2]_C_int) (err error)
31 func Pipe(p []int) (err error) {
32     if len(p) != 2 {
33         return EINVAL
34     }
35     var pp [2]_C_int
36     err = pipe(&pp)
37     p[0] = int(pp[0])
38     p[1] = int(pp[1])
39     return
40 }
41
```

```

42 //sys utimes(path string, times *[2]Timeval) (err error)
43 func Utimes(path string, tv []Timeval) (err error) {
44     if len(tv) != 2 {
45         return EINVAL
46     }
47     return utimes(path, (*[2]Timeval)(unsafe.Pointer(&tv
48 })
49
50 //sys futimesat(dirfd int, path *byte, times *[2]Timeval)
51 func Futimesat(dirfd int, path string, tv []Timeval) (err er
52     if len(tv) != 2 {
53         return EINVAL
54     }
55     return futimesat(dirfd, StringBytePtr(path), (*[2]Ti
56 })
57
58 func Futimes(fd int, tv []Timeval) (err error) {
59     // Believe it or not, this is the best we can do on
60     // (and is what glibc does).
61     return Utimes("/proc/self/fd/"+itoa(fd), tv)
62 }
63
64 const ImplementsGetwd = true
65
66 //sys Getcwd(buf []byte) (n int, err error)
67 func Getwd() (wd string, err error) {
68     var buf [PathMax]byte
69     n, err := Getcwd(buf[0:])
70     if err != nil {
71         return "", err
72     }
73     // Getcwd returns the number of bytes written to buf
74     if n < 1 || n > len(buf) || buf[n-1] != 0 {
75         return "", EINVAL
76     }
77     return string(buf[0 : n-1]), nil
78 }
79
80 func Getgroups() (gids []int, err error) {
81     n, err := getgroups(0, nil)
82     if err != nil {
83         return nil, err
84     }
85     if n == 0 {
86         return nil, nil
87     }
88
89     // Sanity check group count. Max is 1<<16 on Linux.
90     if n < 0 || n > 1<<20 {
91         return nil, EINVAL

```

```

92         }
93
94         a := make([]_Gid_t, n)
95         n, err = getgroups(n, &a[0])
96         if err != nil {
97             return nil, err
98         }
99         gids = make([]int, n)
100        for i, v := range a[0:n] {
101            gids[i] = int(v)
102        }
103        return
104    }
105
106    func Setgroups(gids []int) (err error) {
107        if len(gids) == 0 {
108            return setgroups(0, nil)
109        }
110
111        a := make([]_Gid_t, len(gids))
112        for i, v := range gids {
113            a[i] = _Gid_t(v)
114        }
115        return setgroups(len(a), &a[0])
116    }
117
118    type WaitStatus uint32
119
120    // Wait status is 7 bits at bottom, either 0 (exited),
121    // 0x7F (stopped), or a signal number that caused an exit.
122    // The 0x80 bit is whether there was a core dump.
123    // An extra number (exit code, signal causing a stop)
124    // is in the high bits. At least that's the idea.
125    // There are various irregularities. For example, the
126    // "continued" status is 0xFFFF, distinguishing itself
127    // from stopped via the core dump bit.
128
129    const (
130        mask      = 0x7F
131        core      = 0x80
132        exited   = 0x00
133        stopped   = 0x7F
134        shift     = 8
135    )
136
137    func (w WaitStatus) Exited() bool { return w&mask == exited
138
139    func (w WaitStatus) Signaled() bool { return w&mask != stopp
140

```

```

141 func (w WaitStatus) Stopped() bool { return w&0xFF == stoppe
142
143 func (w WaitStatus) Continued() bool { return w == 0xFFFF }
144
145 func (w WaitStatus) CoreDump() bool { return w.Signed() &&
146
147 func (w WaitStatus) ExitStatus() int {
148     if !w.Exited() {
149         return -1
150     }
151     return int(w>>shift) & 0xFF
152 }
153
154 func (w WaitStatus) Signal() Signal {
155     if !w.Signed() {
156         return -1
157     }
158     return Signal(w & mask)
159 }
160
161 func (w WaitStatus) StopSignal() Signal {
162     if !w.Stopped() {
163         return -1
164     }
165     return Signal(w>>shift) & 0xFF
166 }
167
168 func (w WaitStatus) TrapCause() int {
169     if w.StopSignal() != SIGTRAP {
170         return -1
171     }
172     return int(w>>shift) >> 8
173 }
174
175 //sys wait4(pid int, wstatus *_C_int, options int, rusage
176 func Wait4(pid int, wstatus *WaitStatus, options int, rusage
177     var status _C_int
178     wpid, err = wait4(pid, &status, options, rusage)
179     if wstatus != nil {
180         *wstatus = WaitStatus(status)
181     }
182     return
183 }
184
185 func Mknod(path string, mode uint32) (err error) {
186     return Mknod(path, mode|S_IFIFO, 0)
187 }
188
189 // For testing: clients can set this flag to force

```

```

190 // creation of IPv6 sockets to return EAFNOSUPPORT.
191 var SocketDisableIPv6 bool
192
193 type Sockaddr interface {
194     sockaddr() (ptr uintptr, len _Socklen, err error) //
195 }
196
197 type SockaddrInet4 struct {
198     Port int
199     Addr [4]byte
200     raw RawSockaddrInet4
201 }
202
203 func (sa *SockaddrInet4) sockaddr() (uintptr, _Socklen, error) {
204     if sa.Port < 0 || sa.Port > 0xFFFF {
205         return 0, 0, EINVAL
206     }
207     sa.raw.Family = AF_INET
208     p := (*[2]byte)(unsafe.Pointer(&sa.raw.Port))
209     p[0] = byte(sa.Port >> 8)
210     p[1] = byte(sa.Port)
211     for i := 0; i < len(sa.Addr); i++ {
212         sa.raw.Addr[i] = sa.Addr[i]
213     }
214     return uintptr(unsafe.Pointer(&sa.raw)), SizeofSocka
215 }
216
217 type SockaddrInet6 struct {
218     Port int
219     ZoneId uint32
220     Addr [16]byte
221     raw RawSockaddrInet6
222 }
223
224 func (sa *SockaddrInet6) sockaddr() (uintptr, _Socklen, error) {
225     if sa.Port < 0 || sa.Port > 0xFFFF {
226         return 0, 0, EINVAL
227     }
228     sa.raw.Family = AF_INET6
229     p := (*[2]byte)(unsafe.Pointer(&sa.raw.Port))
230     p[0] = byte(sa.Port >> 8)
231     p[1] = byte(sa.Port)
232     sa.raw.Scope_id = sa.ZoneId
233     for i := 0; i < len(sa.Addr); i++ {
234         sa.raw.Addr[i] = sa.Addr[i]
235     }
236     return uintptr(unsafe.Pointer(&sa.raw)), SizeofSocka
237 }
238
239 type SockaddrUnix struct {

```

```

240         Name string
241         raw RawSockaddrUnix
242     }
243
244     func (sa *SockaddrUnix) sockaddr() (uintptr, _Socklen, error)
245         name := sa.Name
246         n := len(name)
247         if n >= len(sa.raw.Path) || n == 0 {
248             return 0, 0, EINVAL
249         }
250         sa.raw.Family = AF_UNIX
251         for i := 0; i < n; i++ {
252             sa.raw.Path[i] = int8(name[i])
253         }
254         // length is family (uint16), name, NUL.
255         sl := 2 + _Socklen(n) + 1
256         if sa.raw.Path[0] == '@' {
257             sa.raw.Path[0] = 0
258             // Don't count trailing NUL for abstract add
259             sl--
260         }
261
262         return uintptr(unsafe.Pointer(&sa.raw)), sl, nil
263     }
264
265     type SockaddrLinklayer struct {
266         Protocol uint16
267         Ifindex   int
268         Hatype    uint16
269         Pktttype  uint8
270         Halen     uint8
271         Addr      [8]byte
272         raw      RawSockaddrLinklayer
273     }
274
275     func (sa *SockaddrLinklayer) sockaddr() (uintptr, _Socklen,
276         if sa.Ifindex < 0 || sa.Ifindex > 0x7fffffff {
277             return 0, 0, EINVAL
278         }
279         sa.raw.Family = AF_PACKET
280         sa.raw.Protocol = sa.Protocol
281         sa.raw.Ifindex = int32(sa.Ifindex)
282         sa.raw.Hatype = sa.Hatype
283         sa.raw.Pktttype = sa.Pktttype
284         sa.raw.Halen = sa.Halen
285         for i := 0; i < len(sa.Addr); i++ {
286             sa.raw.Addr[i] = sa.Addr[i]
287         }
288         return uintptr(unsafe.Pointer(&sa.raw)), SizeofSocka

```

```

289 }
290
291 type SockaddrNetlink struct {
292     Family uint16
293     Pad     uint16
294     Pid     uint32
295     Groups  uint32
296     raw     RawSockaddrNetlink
297 }
298
299 func (sa *SockaddrNetlink) sockaddr() (uintptr, _Socklen, error) {
300     sa.raw.Family = AF_NETLINK
301     sa.raw.Pad = sa.Pad
302     sa.raw.Pid = sa.Pid
303     sa.raw.Groups = sa.Groups
304     return uintptr(unsafe.Pointer(&sa.raw)), SizeofSockaddrNetlink, nil
305 }
306
307 func anyToSockaddr(rsa *RawSockaddrAny) (Sockaddr, error) {
308     switch rsa.Addr.Family {
309     case AF_NETLINK:
310         pp := (*RawSockaddrNetlink)(unsafe.Pointer(rsa))
311         sa := new(SockaddrNetlink)
312         sa.Family = pp.Family
313         sa.Pad = pp.Pad
314         sa.Pid = pp.Pid
315         sa.Groups = pp.Groups
316         return sa, nil
317
318     case AF_PACKET:
319         pp := (*RawSockaddrLinklayer)(unsafe.Pointer(rsa))
320         sa := new(SockaddrLinklayer)
321         sa.Protocol = pp.Protocol
322         sa.Ifindex = int(pp.Ifindex)
323         sa.Hatype = pp.Hatype
324         sa.Pkttype = pp.Pkttype
325         sa.Halen = pp.Halen
326         for i := 0; i < len(sa.Addr); i++ {
327             sa.Addr[i] = pp.Addr[i]
328         }
329         return sa, nil
330
331     case AF_UNIX:
332         pp := (*RawSockaddrUnix)(unsafe.Pointer(rsa))
333         sa := new(SockaddrUnix)
334         if pp.Path[0] == 0 {
335             // "Abstract" Unix domain socket.
336             // Rewrite leading NUL as @ for text
337             // (This is the standard convention.

```

```

338         // Not friendly to overwrite in plac
339         // but the callers below don't care.
340         pp.Path[0] = '@'
341     }
342
343     // Assume path ends at NUL.
344     // This is not technically the Linux semanti
345     // abstract Unix domain sockets--they are su
346     // to be uninterpreted fixed-size binary blo
347     // everyone uses this convention.
348     n := 0
349     for n < len(pp.Path) && pp.Path[n] != 0 {
350         n++
351     }
352     bytes := (*[10000]byte)(unsafe.Pointer(&pp.P
353     sa.Name = string(bytes)
354     return sa, nil
355
356     case AF_INET:
357         pp := (*RawSockaddrInet4)(unsafe.Pointer(rsa
358         sa := new(SockaddrInet4)
359         p := (*[2]byte)(unsafe.Pointer(&pp.Port))
360         sa.Port = int(p[0])<<8 + int(p[1])
361         for i := 0; i < len(sa.Addr); i++ {
362             sa.Addr[i] = pp.Addr[i]
363         }
364         return sa, nil
365
366     case AF_INET6:
367         pp := (*RawSockaddrInet6)(unsafe.Pointer(rsa
368         sa := new(SockaddrInet6)
369         p := (*[2]byte)(unsafe.Pointer(&pp.Port))
370         sa.Port = int(p[0])<<8 + int(p[1])
371         sa.ZoneId = pp.Scope_id
372         for i := 0; i < len(sa.Addr); i++ {
373             sa.Addr[i] = pp.Addr[i]
374         }
375         return sa, nil
376     }
377     return nil, EAFNOSUPPORT
378 }
379
380 func Accept(fd int) (nfd int, sa Sockaddr, err error) {
381     var rsa RawSockaddrAny
382     var len _Socklen = SizeofSockaddrAny
383     nfd, err = accept(fd, &rsa, &len)
384     if err != nil {
385         return
386     }
387     sa, err = anyToSockaddr(&rsa)

```

```

388         if err != nil {
389             Close(nfd)
390             nfd = 0
391         }
392         return
393     }
394
395     func Getsockname(fd int) (sa Sockaddr, err error) {
396         var rsa RawSockaddrAny
397         var len _Socklen = SizeofSockaddrAny
398         if err = getsockname(fd, &rsa, &len); err != nil {
399             return
400         }
401         return anyToSockaddr(&rsa)
402     }
403
404     func Getpeername(fd int) (sa Sockaddr, err error) {
405         var rsa RawSockaddrAny
406         var len _Socklen = SizeofSockaddrAny
407         if err = getpeername(fd, &rsa, &len); err != nil {
408             return
409         }
410         return anyToSockaddr(&rsa)
411     }
412
413     func Bind(fd int, sa Sockaddr) (err error) {
414         ptr, n, err := sa.sockaddr()
415         if err != nil {
416             return err
417         }
418         return bind(fd, ptr, n)
419     }
420
421     func Connect(fd int, sa Sockaddr) (err error) {
422         ptr, n, err := sa.sockaddr()
423         if err != nil {
424             return err
425         }
426         return connect(fd, ptr, n)
427     }
428
429     func Socket(domain, typ, proto int) (fd int, err error) {
430         if domain == AF_INET6 && SocketDisableIPv6 {
431             return -1, EAFNOSUPPORT
432         }
433         fd, err = socket(domain, typ, proto)
434         return
435     }
436

```

```

437 func Socketpair(domain, typ, proto int) (fd [2]int, err error) {
438     err = socketpair(domain, typ, proto, &fd)
439     return
440 }
441
442 func GetsockoptInt(fd, level, opt int) (value int, err error) {
443     var n int32
444     vallen := _Socklen(4)
445     err = getsockopt(fd, level, opt, uintptr(unsafe.Pointer(&n)))
446     return int(n), err
447 }
448
449 func GetsockoptInet4Addr(fd, level, opt int) (value [4]byte, err error) {
450     vallen := _Socklen(4)
451     err = getsockopt(fd, level, opt, uintptr(unsafe.Pointer(&value)))
452     return value, err
453 }
454
455 func GetsockoptIPMreq(fd, level, opt int) (*IPMreq, error) {
456     var value IPMreq
457     vallen := _Socklen(SizeofIPMreq)
458     err := getsockopt(fd, level, opt, uintptr(unsafe.Pointer(&value)))
459     return &value, err
460 }
461
462 func GetsockoptIPMreqn(fd, level, opt int) (*IPMreqn, error) {
463     var value IPMreqn
464     vallen := _Socklen(SizeofIPMreqn)
465     err := getsockopt(fd, level, opt, uintptr(unsafe.Pointer(&value)))
466     return &value, err
467 }
468
469 func GetsockoptIPv6Mreq(fd, level, opt int) (*IPv6Mreq, error) {
470     var value IPv6Mreq
471     vallen := _Socklen(SizeofIPv6Mreq)
472     err := getsockopt(fd, level, opt, uintptr(unsafe.Pointer(&value)))
473     return &value, err
474 }
475
476 func SetsockoptInt(fd, level, opt int, value int) (err error) {
477     var n = int32(value)
478     return setsockopt(fd, level, opt, uintptr(unsafe.Pointer(&n)))
479 }
480
481 func SetsockoptInet4Addr(fd, level, opt int, value [4]byte) (err error) {
482     return setsockopt(fd, level, opt, uintptr(unsafe.Pointer(&value)))
483 }
484
485 func SetsockoptTimeval(fd, level, opt int, tv *Timeval) (err error) {

```

```

486         return setsockopt(fd, level, opt, uintptr(unsafe.Poi
487     })
488
489     func SetsockoptLinger(fd, level, opt int, l *Linger) (err er
490         return setsockopt(fd, level, opt, uintptr(unsafe.Poi
491     })
492
493     func SetsockoptIPMreq(fd, level, opt int, mreq *IPMreq) (err
494         return setsockopt(fd, level, opt, uintptr(unsafe.Poi
495     })
496
497     func SetsockoptIPMreqn(fd, level, opt int, mreq *IPMreqn) (e
498         return setsockopt(fd, level, opt, uintptr(unsafe.Poi
499     })
500
501     func SetsockoptIPv6Mreq(fd, level, opt int, mreq *IPv6Mreq)
502         return setsockopt(fd, level, opt, uintptr(unsafe.Poi
503     })
504
505     func SetsockoptString(fd, level, opt int, s string) (err err
506         return setsockopt(fd, level, opt, uintptr(unsafe.Poi
507     })
508
509     func Recvfrom(fd int, p []byte, flags int) (n int, from Sock
510         var rsa RawSockaddrAny
511         var len _Socklen = SizeofSockaddrAny
512         if n, err = recvfrom(fd, p, flags, &rsa, &len); err
513             return
514         }
515         from, err = anyToSockaddr(&rsa)
516         return
517     }
518
519     func Sendto(fd int, p []byte, flags int, to Sockaddr) (err e
520         ptr, n, err := to.sockaddr()
521         if err != nil {
522             return err
523         }
524         return sendto(fd, p, flags, ptr, n)
525     }
526
527     func Recvmsg(fd int, p, oob []byte, flags int) (n, oobn int,
528         var msg MsgHdr
529         var rsa RawSockaddrAny
530         msg.Name = (*byte)(unsafe.Pointer(&rsa))
531         msg.Namelen = uint32(SizeofSockaddrAny)
532         var iov Iovec
533         if len(p) > 0 {
534             iov.Base = (*byte)(unsafe.Pointer(&p[0]))
535             iov.SetLen(len(p))

```

```

536     }
537     var dummy byte
538     if len(oob) > 0 {
539         // receive at least one normal byte
540         if len(p) == 0 {
541             iov.Base = &dummy
542             iov.SetLen(1)
543         }
544         msg.Control = (*byte)(unsafe.Pointer(&oob[0])
545         msg.SetControllen(len(oob))
546     }
547     msg.Iov = &iov
548     msg.Iovlen = 1
549     if n, err = recvmmsg(fd, &msg, flags); err != nil {
550         return
551     }
552     oobn = int(msg.Controllen)
553     recvflags = int(msg.Flags)
554     // source address is only specified if the socket is
555     if rsa.Addr.Family != AF_UNSPEC {
556         from, err = anyToSockaddr(&rsa)
557     }
558     return
559 }
560
561 func Sendmmsg(fd int, p, oob []byte, to Sockaddr, flags int)
562     var ptr uintptr
563     var salen _Socklen
564     if to != nil {
565         var err error
566         ptr, salen, err = to.sockaddr()
567         if err != nil {
568             return err
569         }
570     }
571     var msg MsgHdr
572     msg.Name = (*byte)(unsafe.Pointer(ptr))
573     msg.Namelen = uint32(salen)
574     var iov Iovec
575     if len(p) > 0 {
576         iov.Base = (*byte)(unsafe.Pointer(&p[0]))
577         iov.SetLen(len(p))
578     }
579     var dummy byte
580     if len(oob) > 0 {
581         // send at least one normal byte
582         if len(p) == 0 {
583             iov.Base = &dummy
584             iov.SetLen(1)

```

```

585         }
586         msg.Control = (*byte)(unsafe.Pointer(&oob[0])
587         msg.SetControlLen(len(oob))
588     }
589     msg.Iov = &iiov
590     msg.IovLen = 1
591     if err = sendmsg(fd, &msg, flags); err != nil {
592         return
593     }
594     return
595 }
596
597 // BindToDevice binds the socket associated with fd to device
598 func BindToDevice(fd int, device string) (err error) {
599     return SetsockoptString(fd, SOL_SOCKET, SO_BINDTODEVICE, device)
600 }
601
602 //sys ptrace(request int, pid int, addr uintptr, data uint) (err error)
603
604 func ptracePeek(req int, pid int, addr uintptr, out []byte) (err error) {
605     // The peek requests are machine-size oriented, so we need to
606     // to retrieve arbitrary-length data.
607
608     // The ptrace syscall differs from glibc's ptrace.
609     // Peek returns the word in *data, not as the return value.
610
611     var buf [sizeofPtr]byte
612
613     // Leading edge. PEEKTEXT/PEEKDATA don't require all
614     // access (PEEKUSER warns that it might), but if we
615     // align our reads, we might straddle an unmapped page
616     // boundary and not get the bytes leading up to the
617     // boundary.
618     n := 0
619     if addr%sizeofPtr != 0 {
620         err = ptrace(req, pid, addr-addr%sizeofPtr, 0)
621         if err != nil {
622             return 0, err
623         }
624         n += copy(out, buf[addr%sizeofPtr:])
625         out = out[n:]
626     }
627
628     // Remainder.
629     for len(out) > 0 {
630         // We use an internal buffer to guarantee all
631         // It's not documented if this is necessary,
632         err = ptrace(req, pid, addr+uintptr(n), uint)
633         if err != nil {

```

```

634             return n, err
635         }
636         copied := copy(out, buf[0:])
637         n += copied
638         out = out[copied:]
639     }
640     return n, nil
641 }
642 }
643
644 func PtracePeekText(pid int, addr uintptr, out []byte) (count int, err error) {
645     return ptracePeek(PTRACE_PEEKTEXT, pid, addr, out)
646 }
647
648 func PtracePeekData(pid int, addr uintptr, out []byte) (count int, err error) {
649     return ptracePeek(PTRACE_PEEKDATA, pid, addr, out)
650 }
651
652 func ptracePoke(pokeReq int, peekReq int, pid int, addr uint, data []byte) (count int, err error) {
653     // As for ptracePeek, we need to align our accesses
654     // with the possibility of straddling an invalid page
655
656     // Leading edge.
657     n := 0
658     if addr%sizeofPtr != 0 {
659         var buf [sizeofPtr]byte
660         err = ptrace(peekReq, pid, addr-addr%sizeofPtr, &buf)
661         if err != nil {
662             return 0, err
663         }
664         n += copy(buf[addr%sizeofPtr:], data)
665         word := *((*uintptr)(unsafe.Pointer(&buf[0])))
666         err = ptrace(pokeReq, pid, addr-addr%sizeofPtr, word)
667         if err != nil {
668             return 0, err
669         }
670         data = data[n:]
671     }
672
673     // Interior.
674     for len(data) > sizeofPtr {
675         word := *((*uintptr)(unsafe.Pointer(&data[0])))
676         err = ptrace(pokeReq, pid, addr+uintptr(n), word)
677         if err != nil {
678             return n, err
679         }
680         n += sizeofPtr
681         data = data[sizeofPtr:]
682     }
683 }

```

```

684         // Trailing edge.
685         if len(data) > 0 {
686             var buf [sizeofPtr]byte
687             err = ptrace(peekReq, pid, addr+uintptr(n),
688             if err != nil {
689                 return n, err
690             }
691             copy(buf[0:], data)
692             word := *((*uintptr)(unsafe.Pointer(&buf[0])))
693             err = ptrace(pokeReq, pid, addr+uintptr(n),
694             if err != nil {
695                 return n, err
696             }
697             n += len(data)
698         }
699
700     return n, nil
701 }
702
703 func PtracePokeText(pid int, addr uintptr, data []byte) (cou
704     return ptracePoke(PTRACE_POKETEXT, PTRACE_PEEKTEXT,
705 }
706
707 func PtracePokeData(pid int, addr uintptr, data []byte) (cou
708     return ptracePoke(PTRACE_POKEDATA, PTRACE_PEEKDATA,
709 }
710
711 func PtraceGetRegs(pid int, regsout *PtraceRegs) (err error)
712     return ptrace(PTRACE_GETREGS, pid, 0, uintptr(unsafe
713 }
714
715 func PtraceSetRegs(pid int, regs *PtraceRegs) (err error) {
716     return ptrace(PTRACE_SETREGS, pid, 0, uintptr(unsafe
717 }
718
719 func PtraceSetOptions(pid int, options int) (err error) {
720     return ptrace(PTRACE_SETOPTIONS, pid, 0, uintptr(opt
721 }
722
723 func PtraceGetEventMsg(pid int) (msg uint, err error) {
724     var data _C_long
725     err = ptrace(PTRACE_GETEVENTMSG, pid, 0, uintptr(uns
726     msg = uint(data)
727     return
728 }
729
730 func PtraceCont(pid int, signal int) (err error) {
731     return ptrace(PTRACE_CONT, pid, 0, uintptr(signal))
732 }

```

```

733
734 func PtraceSingleStep(pid int) (err error) { return ptrace(P
735
736 func PtraceAttach(pid int) (err error) { return ptrace(PTRAC
737
738 func PtraceDetach(pid int) (err error) { return ptrace(PTRAC
739
740 //sys reboot(magic1 uint, magic2 uint, cmd int, arg string
741 func Reboot(cmd int) (err error) {
742     return reboot(LINUX_REBOOT_MAGIC1, LINUX_REBOOT_MAGI
743 }
744
745 func clen(n []byte) int {
746     for i := 0; i < len(n); i++ {
747         if n[i] == 0 {
748             return i
749         }
750     }
751     return len(n)
752 }
753
754 func ReadDirent(fd int, buf []byte) (n int, err error) {
755     return Getdents(fd, buf)
756 }
757
758 func ParseDirent(buf []byte, max int, names []string) (consu
759     origlen := len(buf)
760     count = 0
761     for max != 0 && len(buf) > 0 {
762         dirent := (*Dirent)(unsafe.Pointer(&buf[0]))
763         buf = buf[dirent.Reclen:]
764         if dirent.Ino == 0 { // File absent in direc
765             continue
766         }
767         bytes := (*[10000]byte)(unsafe.Pointer(&dire
768         var name = string(bytes[0:clen(bytes[:])])
769         if name == "." || name == ".." { // Useless
770             continue
771         }
772         max--
773         count++
774         names = append(names, name)
775     }
776     return origlen - len(buf), count, names
777 }
778
779 //sys mount(source string, target string, fstype string, f
780 func Mount(source string, target string, fstype string, flag
781     // Certain file systems get rather angry and EINVAL

```

```

782         // them an empty string of data, rather than NULL.
783         if data == "" {
784             return mount(source, target, fstype, flags,
785                 }
786         return mount(source, target, fstype, flags, StringBy
787     }
788
789     // Sendto
790     // Recvfrom
791     // Socketpair
792
793     /*
794     * Direct access
795     */
796     //sys  Access(path string, mode uint32) (err error)
797     //sys  Acct(path string) (err error)
798     //sys  Adjtimex(buf *Timex) (state int, err error)
799     //sys  Chdir(path string) (err error)
800     //sys  Chmod(path string, mode uint32) (err error)
801     //sys  Chroot(path string) (err error)
802     //sys  Close(fd int) (err error)
803     //sys  Creat(path string, mode uint32) (fd int, err error)
804     //sysnb Dup(oldfd int) (fd int, err error)
805     //sysnb Dup2(oldfd int, newfd int) (err error)
806     //sysnb EpollCreate(size int) (fd int, err error)
807     //sysnb EpollCreate1(flag int) (fd int, err error)
808     //sysnb EpollCtl(epfd int, op int, fd int, event *EpollEvent
809     //sys  EpollWait(epfd int, events []EpollEvent, msec int) (
810     //sys  Exit(code int) = SYS_EXIT_GROUP
811     //sys  Faccessat(dirfd int, path string, mode uint32, flags
812     //sys  Fallocate(fd int, mode uint32, off int64, len int64)
813     //sys  Fchdir(fd int) (err error)
814     //sys  Fchmod(fd int, mode uint32) (err error)
815     //sys  Fchmodat(dirfd int, path string, mode uint32, flags
816     //sys  Fchownat(dirfd int, path string, uid int, gid int, f
817     //sys  fcntl(fd int, cmd int, arg int) (val int, err error)
818     //sys  Fdatasync(fd int) (err error)
819     //sys  Flock(fd int, how int) (err error)
820     //sys  Fsync(fd int) (err error)
821     //sys  Getdents(fd int, buf []byte) (n int, err error) = SY
822     //sysnb Getpgid(pid int) (pgid int, err error)
823     //sysnb Getpgrp() (pid int)
824     //sysnb Getpid() (pid int)
825     //sysnb Getppid() (ppid int)
826     //sysnb Getrlimit(resource int, rlim *Rlimit) (err error)
827     //sysnb Getrusage(who int, rusage *Rusage) (err error)
828     //sysnb Gettid() (tid int)
829     //sys  InotifyAddWatch(fd int, pathname string, mask uint32
830     //sysnb InotifyInit() (fd int, err error)
831     //sysnb InotifyInit1(flags int) (fd int, err error)

```

```

832 //sysnb InotifyRmWatch(fd int, watchdesc uint32) (success in
833 //sysnb Kill(pid int, sig Signal) (err error)
834 //sys Klogctl(typ int, buf []byte) (n int, err error) = SY
835 //sys Link(oldpath string, newpath string) (err error)
836 //sys Mkdir(path string, mode uint32) (err error)
837 //sys Mkdirat(dirfd int, path string, mode uint32) (err er
838 //sys Mknod(path string, mode uint32, dev int) (err error)
839 //sys Mknodat(dirfd int, path string, mode uint32, dev int
840 //sys Nanosleep(time *Timespec, leftover *Timespec) (err e
841 //sys Pause() (err error)
842 //sys PivotRoot(newroot string, putold string) (err error)
843 //sys Read(fd int, p []byte) (n int, err error)
844 //sys Readlink(path string, buf []byte) (n int, err error)
845 //sys Rename(oldpath string, newpath string) (err error)
846 //sys Renameat.olddirfd int, oldpath string, newdirfd int,
847 //sys Rmdir(path string) (err error)
848 //sys Setdomainname(p []byte) (err error)
849 //sys Sethostname(p []byte) (err error)
850 //sysnb Setpgid(pid int, pgid int) (err error)
851 //sysnb Setrlimit(resource int, rlim *Rlimit) (err error)
852 //sysnb Setsid() (pid int, err error)
853 //sysnb Settimeofday(tv *Timeval) (err error)
854 //sysnb Setuid(uid int) (err error)
855 //sys Symlink(oldpath string, newpath string) (err error)
856 //sys Sync()
857 //sysnb Sysinfo(info *Sysinfo_t) (err error)
858 //sys Tee(rfd int, wfd int, len int, flags int) (n int64,
859 //sysnb Tgkill(tgid int, tid int, sig Signal) (err error)
860 //sysnb Times(tms *Tms) (ticks uintptr, err error)
861 //sysnb Umask(mask int) (oldmask int)
862 //sysnb Uname(buf *Utsname) (err error)
863 //sys Unlink(path string) (err error)
864 //sys Unlinkat(dirfd int, path string) (err error)
865 //sys Unmount(target string, flags int) (err error) = SYS_
866 //sys Unshare(flags int) (err error)
867 //sys Ustat(dev int, ubuf *Ustat_t) (err error)
868 //sys Utime(path string, buf *Utimbuf) (err error)
869 //sys Write(fd int, p []byte) (n int, err error)
870 //sys exitThread(code int) (err error) = SYS_EXIT
871 //sys read(fd int, p *byte, np int) (n int, err error)
872 //sys write(fd int, p *byte, np int) (n int, err error)
873
874 // mmap varies by architecture; see syscall_linux*.go.
875 //sys munmap(addr uintptr, length uintptr) (err error)
876
877 var mapper = &mapper{
878     active: make(map[*byte][]byte),
879     mmap:   mmap,
880     munmap: munmap,

```

```
881 }
882
883 func Mmap(fd int, offset int64, length int, prot int, flags
884         return mapper.Mmap(fd, offset, length, prot, flags)
885 }
886
887 func Munmap(b []byte) (err error) {
888     return mapper.Munmap(b)
889 }
890
891 //sys  Madvise(b []byte, advice int) (err error)
892 //sys  Mprotect(b []byte, prot int) (err error)
893 //sys  Mlock(b []byte) (err error)
894 //sys  Munlock(b []byte) (err error)
895 //sys  Mlockall(flags int) (err error)
896 //sys  Munlockall() (err error)
897
898 /*
899  * Unimplemented
900  */
901 // AddKey
902 // AfsSyscall
903 // Alarm
904 // ArchPrctl
905 // Brk
906 // Capget
907 // Capset
908 // ClockGetres
909 // ClockGettime
910 // ClockNanosleep
911 // ClockSettime
912 // Clone
913 // CreateModule
914 // DeleteModule
915 // EpollCtlOld
916 // EpollPwait
917 // EpollWaitOld
918 // Eventfd
919 // Execve
920 // Fadvise64
921 // Fgetxattr
922 // Flistxattr
923 // Fork
924 // Fremovexattr
925 // Fsetxattr
926 // Futex
927 // GetKernelSyms
928 // GetMempolicy
929 // GetRobustList
```

```
930 // GetThreadArea
931 // Getitimer
932 // Getpmsg
933 // Getpriority
934 // Getxattr
935 // IoCancel
936 // IoDestroy
937 // IoGetevents
938 // IoSetup
939 // IoSubmit
940 // Ioctl
941 // IoprioGet
942 // IoprioSet
943 // KexecLoad
944 // Keyctl
945 // Lgetxattr
946 // Listxattr
947 // Llistxattr
948 // LookupDcookie
949 // Lremovexattr
950 // Lsetxattr
951 // Mbind
952 // MigratePages
953 // Mincore
954 // ModifyLdt
955 // Mount
956 // MovePages
957 // Mprotect
958 // MqGetsetattr
959 // MqNotify
960 // MqOpen
961 // MqTimedreceive
962 // MqTimedsend
963 // MqUnlink
964 // Mremap
965 // Msgctl
966 // Msgget
967 // Msgrcv
968 // Msgsnd
969 // Msync
970 // Newfstatat
971 // Nfsservctl
972 // Personality
973 // Poll
974 // Ppoll
975 // Prctl
976 // Pselect6
977 // Ptrace
978 // Putpmsg
979 // QueryModule
```

```
980 // Quotactl
981 // Readahead
982 // Readv
983 // RemapFilePages
984 // Removexattr
985 // RequestKey
986 // RestartSyscall
987 // RtSigaction
988 // RtSigpending
989 // RtSigprocmask
990 // RtSigqueueinfo
991 // RtSigreturn
992 // RtSigsuspend
993 // RtSigtimedwait
994 // SchedGetPriorityMax
995 // SchedGetPriorityMin
996 // SchedGetaffinity
997 // SchedGetparam
998 // SchedGetscheduler
999 // SchedRrGetInterval
1000 // SchedSetaffinity
1001 // SchedSetparam
1002 // SchedYield
1003 // Security
1004 // Semctl
1005 // Semget
1006 // Semop
1007 // Sementimedop
1008 // SetMempolicy
1009 // SetRobustList
1010 // SetThreadArea
1011 // SetTidAddress
1012 // Setpriority
1013 // Setxattr
1014 // Shmat
1015 // Shmctl
1016 // Shmdt
1017 // Shmget
1018 // Sigaltstack
1019 // Signalfd
1020 // Swapoff
1021 // Swapon
1022 // Sysfs
1023 // TimerCreate
1024 // TimerDelete
1025 // TimerGetoverrun
1026 // TimerGettime
1027 // TimerSettime
1028 // Timerfd
```

```
1029 // Tkill (obsolete)
1030 // Tuxcall
1031 // Umount2
1032 // Uselib
1033 // Utimensat
1034 // Vfork
1035 // Vhangup
1036 // Vmsplice
1037 // Vserver
1038 // Waitid
1039 // Writev
1040 // _Sysctl
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/syscall_linux_amd64.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package syscall
6
7 //sys  Chown(path string, uid int, gid int) (err error)
8 //sys  Fchown(fd int, uid int, gid int) (err error)
9 //sys  Fstat(fd int, stat *Stat_t) (err error)
10 //sys  Fstatfs(fd int, buf *Statfs_t) (err error)
11 //sys  Ftruncate(fd int, length int64) (err error)
12 //sysnb Getegid() (egid int)
13 //sysnb Geteuid() (euid int)
14 //sysnb Getgid() (gid int)
15 //sysnb Getuid() (uid int)
16 //sys  Ioperm(from int, num int, on int) (err error)
17 //sys  Iopl(level int) (err error)
18 //sys  Lchown(path string, uid int, gid int) (err error)
19 //sys  Listen(s int, n int) (err error)
20 //sys  Lstat(path string, stat *Stat_t) (err error)
21 //sys  Pread(fd int, p []byte, offset int64) (n int, err error)
22 //sys  Pwrite(fd int, p []byte, offset int64) (n int, err error)
23 //sys  Seek(fd int, offset int64, whence int) (off int64, err error)
24 //sys  Select(nfd int, r *FdSet, w *FdSet, e *FdSet, timeout *Timeval) (n int)
25 //sys  Sendfile(outfd int, infd int, offset *int64, count int) (err error)
26 //sys  Setfsgid(gid int) (err error)
27 //sys  Setfsuid(uid int) (err error)
28 //sysnb Setgid(gid int) (err error)
29 //sysnb Setregid(rgid int, egid int) (err error)
30 //sysnb Setresgid(rgid int, egid int, sgid int) (err error)
31 //sysnb Setresuid(ruid int, euid int, suid int) (err error)
32 //sysnb Setreuid(ruid int, euid int) (err error)
33 //sys  Shutdown(fd int, how int) (err error)
34 //sys  Splice(rfd int, roff *int64, wfd int, woff *int64, len int) (err error)
35 //sys  Stat(path string, stat *Stat_t) (err error)
36 //sys  Statfs(path string, buf *Statfs_t) (err error)
37 //sys  SyncFileRange(fd int, off int64, n int64, flags int) (err error)
38 //sys  Truncate(path string, length int64) (err error)
39 //sys  accept(s int, rsa *RawSockaddrAny, addrlen *_Socklen) (fd int, rsa RawSockaddrAny, addrlen *_Socklen)
40 //sys  bind(s int, addr uintptr, addrlen _Socklen) (err error)
41 //sys  connect(s int, addr uintptr, addrlen _Socklen) (err error)
```

```

42 //sysnb getgroups(n int, list *_Gid_t) (nn int, err error)
43 //sysnb setgroups(n int, list *_Gid_t) (err error)
44 //sys  getsockopt(s int, level int, name int, val uintptr,
45 //sys  setsockopt(s int, level int, name int, val uintptr,
46 //sysnb socket(domain int, typ int, proto int) (fd int, err
47 //sysnb socketpair(domain int, typ int, proto int, fd *[2]in
48 //sysnb getpeername(fd int, rsa *RawSockaddrAny, addrlen *_S
49 //sysnb getsockname(fd int, rsa *RawSockaddrAny, addrlen *_S
50 //sys  recvfrom(fd int, p []byte, flags int, from *RawSocka
51 //sys  sendto(s int, buf []byte, flags int, to uintptr, add
52 //sys  recvmsg(s int, msg *Msghdr, flags int) (n int, err e
53 //sys  sendmsg(s int, msg *Msghdr, flags int) (err error)
54 //sys  mmap(addr uintptr, length uintptr, prot int, flags i
55
56 func Getpagesize() int { return 4096 }
57
58 func Gettimeofday(tv *Timeval) (err error)
59 func Time(t *Time_t) (tt Time_t, err error)
60
61 func TimespecToNsec(ts Timespec) int64 { return int64(ts.Sec
62
63 func NsecToTimespec(nsec int64) (ts Timespec) {
64     ts.Sec = nsec / 1e9
65     ts.Nsec = nsec % 1e9
66     return
67 }
68
69 func TimevalToNsec(tv Timeval) int64 { return int64(tv.Sec)*
70
71 func NsecToTimeval(nsec int64) (tv Timeval) {
72     nsec += 999 // round up to microsecond
73     tv.Sec = nsec / 1e9
74     tv.Usec = nsec % 1e9 / 1e3
75     return
76 }
77
78 func (r *PtraceRegs) PC() uint64 { return r.Rip }
79
80 func (r *PtraceRegs) SetPC(pc uint64) { r.Rip = pc }
81
82 func (iov *Iovec) SetLen(length int) {
83     iov.Len = uint64(length)
84 }
85
86 func (msghdr *Msghdr) SetControllen(length int) {
87     msghdr.Controllen = uint64(length)
88 }
89
90 func (cmsg *Cmsghdr) SetLen(length int) {
91     cmsg.Len = uint64(length)

```

92 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/syscall_unix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package syscall
8
9 import (
10     "runtime"
11     "sync"
12     "unsafe"
13 )
14
15 var (
16     Stdin  = 0
17     Stdout = 1
18     Stderr = 2
19 )
20
21 const darwinAMD64 = runtime.GOOS == "darwin" && runtime.GOARCH == "amd64"
22
23 func Syscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err syscall.Errno)
24 func Syscall6(trap, a1, a2, a3, a4, a5, a6 uintptr) (r1, r2 uintptr, err syscall.Errno)
25 func RawSyscall(trap, a1, a2, a3 uintptr) (r1, r2 uintptr, err syscall.Errno)
26 func RawSyscall6(trap, a1, a2, a3, a4, a5, a6 uintptr) (r1, r2 uintptr, err syscall.Errno)
27
28 // Mmap manager, for use by operating system-specific implementations
29
30 type mmapper struct {
31     sync.Mutex
32     active map[*byte][*byte] // active mappings; key is l
33     mmap   func(addr, length uintptr, prot, flags, fd int) (uintptr, error)
34     munmap func(addr uintptr, length uintptr) error
35 }
36
37 func (m *mmapper) Mmap(fd int, offset int64, length int, prot int, flags int) (uintptr, error) {
38     if length <= 0 {
39         return nil, EINVAL
40     }
41 }
```

```

42         // Map the requested memory.
43         addr, errno := m.mmap(0, uintptr(length), prot, flag
44         if errno != nil {
45             return nil, errno
46         }
47
48         // Slice memory layout
49         var sl = struct {
50             addr uintptr
51             len  int
52             cap  int
53         }{addr, length, length}
54
55         // Use unsafe to turn sl into a []byte.
56         b := *(*[]byte)(unsafe.Pointer(&sl))
57
58         // Register mapping in m and return it.
59         p := &b[cap(b)-1]
60         m.Lock()
61         defer m.Unlock()
62         m.active[p] = b
63         return b, nil
64     }
65
66     func (m *mmapper) Munmap(data []byte) (err error) {
67         if len(data) == 0 || len(data) != cap(data) {
68             return EINVAL
69         }
70
71         // Find the base of the mapping.
72         p := &data[cap(data)-1]
73         m.Lock()
74         defer m.Unlock()
75         b := m.active[p]
76         if b == nil || &b[0] != &data[0] {
77             return EINVAL
78         }
79
80         // Unmap the memory and update m.
81         if errno := m.munmap(uintptr(unsafe.Pointer(&b[0])),
82             return errno
83         }
84         delete(m.active, p)
85         return nil
86     }
87
88     // An Errno is an unsigned number describing an error condit
89     // It implements the error interface. The zero Errno is by
90     // a non-error, so code to convert from Errno to error shoul
91     //     err = nil

```

```

92 //      if errno != 0 {
93 //          err = errno
94 //      }
95 type Errno uintptr
96
97 func (e Errno) Error() string {
98     if 0 <= int(e) && int(e) < len(errors) {
99         s := errors[e]
100        if s != "" {
101            return s
102        }
103    }
104    return "errno " + itoa(int(e))
105 }
106
107 func (e Errno) Temporary() bool {
108     return e == EINTR || e == EMFILE || e.Timeout()
109 }
110
111 func (e Errno) Timeout() bool {
112     return e == EAGAIN || e == EWOULDBLOCK || e == ETIME
113 }
114
115 // A Signal is a number describing a process signal.
116 // It implements the os.Signal interface.
117 type Signal int
118
119 func (s Signal) Signal() {}
120
121 func (s Signal) String() string {
122     if 0 <= s && int(s) < len(signals) {
123         str := signals[s]
124         if str != "" {
125             return str
126         }
127     }
128     return "signal " + itoa(int(s))
129 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/zerrors_linux_amd64.g

```
1 // mkerrors.sh -m64
2 // MACHINE GENERATED BY THE COMMAND ABOVE; DO NOT EDIT
3
4 // Created by cgo -godefs - DO NOT EDIT
5 // cgo -godefs -- -m64 _const.go
6
7 //line _const.go:1
8 package syscall
9
10 //line _const.go:51
11
12 //line _const.go:50
13 const (
14     AF_ALG                = 0x26
15     AF_APPLETALK          = 0x5
16     AF_ASH                 = 0x12
17     AF_ATMPVC             = 0x8
18     AF_ATMSVC             = 0x14
19     AF_AX25               = 0x3
20     AF_BLUETOOTH          = 0x1f
21     AF_BRIDGE             = 0x7
22     AF_CAIF               = 0x25
23     AF_CAN                = 0x1d
24     AF_DECnet             = 0xc
25     AF_ECONET             = 0x13
26     AF_FILE               = 0x1
27     AF_IEEE802154        = 0x24
28     AF_INET               = 0x2
29     AF_INET6              = 0xa
30     AF_IPX                = 0x4
31     AF_IRDA               = 0x17
32     AF_ISDN               = 0x22
33     AF_IUCV               = 0x20
34     AF_KEY                = 0xf
35     AF_LLC                = 0x1a
36     AF_LOCAL              = 0x1
37     AF_MAX                = 0x27
38     AF_NETBEUI            = 0xd
39     AF_NETLINK           = 0x10
40     AF_NETROM             = 0x6
41     AF_PACKET            = 0x11
```

42	AF_PHONET	= 0x23
43	AF_PPPOX	= 0x18
44	AF_RDS	= 0x15
45	AF_ROSE	= 0xb
46	AF_ROUTE	= 0x10
47	AF_RXRPC	= 0x21
48	AF_SECURITY	= 0xe
49	AF_SNA	= 0x16
50	AF_TIPC	= 0x1e
51	AF_UNIX	= 0x1
52	AF_UNSPEC	= 0x0
53	AF_WANPIPE	= 0x19
54	AF_X25	= 0x9
55	ARPHRD_ADAPT	= 0x108
56	ARPHRD_APPLETLK	= 0x8
57	ARPHRD_ARCNET	= 0x7
58	ARPHRD_ASH	= 0x30d
59	ARPHRD_ATM	= 0x13
60	ARPHRD_AX25	= 0x3
61	ARPHRD_BIF	= 0x307
62	ARPHRD_CHAOS	= 0x5
63	ARPHRD_CISCO	= 0x201
64	ARPHRD_CSLIP	= 0x101
65	ARPHRD_CSLIP6	= 0x103
66	ARPHRD_DDCMP	= 0x205
67	ARPHRD_DLCI	= 0xf
68	ARPHRD_ECONET	= 0x30e
69	ARPHRD_EETHER	= 0x2
70	ARPHRD_ETHER	= 0x1
71	ARPHRD_EUI64	= 0x1b
72	ARPHRD_FCAL	= 0x311
73	ARPHRD_FCFABRIC	= 0x313
74	ARPHRD_FCPL	= 0x312
75	ARPHRD_FCPP	= 0x310
76	ARPHRD_FDDI	= 0x306
77	ARPHRD_FRAD	= 0x302
78	ARPHRD_HDLC	= 0x201
79	ARPHRD_HIPPI	= 0x30c
80	ARPHRD_HWX25	= 0x110
81	ARPHRD_IEEE1394	= 0x18
82	ARPHRD_IEEE802	= 0x6
83	ARPHRD_IEEE80211	= 0x321
84	ARPHRD_IEEE80211_PRISM	= 0x322
85	ARPHRD_IEEE80211_RADIOTAP	= 0x323
86	ARPHRD_IEEE802154	= 0x324
87	ARPHRD_IEEE802154_PHY	= 0x325
88	ARPHRD_IEEE802_TR	= 0x320
89	ARPHRD_INFINIBAND	= 0x20
90	ARPHRD_IPDDP	= 0x309
91	ARPHRD_IPGRE	= 0x30a

92	ARPHRD_IRDA	= 0x30f
93	ARPHRD_LAPB	= 0x204
94	ARPHRD_LOCALTLK	= 0x305
95	ARPHRD_LOOPBACK	= 0x304
96	ARPHRD_METRICOM	= 0x17
97	ARPHRD_NETROM	= 0x0
98	ARPHRD_NONE	= 0xffffe
99	ARPHRD_PIMREG	= 0x30b
100	ARPHRD_PPP	= 0x200
101	ARPHRD_PRONET	= 0x4
102	ARPHRD_RAWHDLC	= 0x206
103	ARPHRD_ROSE	= 0x10e
104	ARPHRD_RSRVD	= 0x104
105	ARPHRD_SIT	= 0x308
106	ARPHRD_SKIP	= 0x303
107	ARPHRD_SLIP	= 0x100
108	ARPHRD_SLIP6	= 0x102
109	ARPHRD_TUNNEL	= 0x300
110	ARPHRD_TUNNEL6	= 0x301
111	ARPHRD_VOID	= 0xfffff
112	ARPHRD_X25	= 0x10f
113	BPF_A	= 0x10
114	BPF_ABS	= 0x20
115	BPF_ADD	= 0x0
116	BPF_ALU	= 0x4
117	BPF_AND	= 0x50
118	BPF_B	= 0x10
119	BPF_DIV	= 0x30
120	BPF_H	= 0x8
121	BPF_IMM	= 0x0
122	BPF_IND	= 0x40
123	BPF_JA	= 0x0
124	BPF_JEQ	= 0x10
125	BPF_JGE	= 0x30
126	BPF_JGT	= 0x20
127	BPF_JMP	= 0x5
128	BPF_JSET	= 0x40
129	BPF_K	= 0x0
130	BPF_LD	= 0x0
131	BPF_LDX	= 0x1
132	BPF_LEN	= 0x80
133	BPF_LSH	= 0x60
134	BPF_MAJOR_VERSION	= 0x1
135	BPF_MAXINSNS	= 0x1000
136	BPF_MEM	= 0x60
137	BPF_MEMWORDS	= 0x10
138	BPF_MINOR_VERSION	= 0x1
139	BPF_MISC	= 0x7
140	BPF_MSH	= 0xa0

141	BPF_MUL	= 0x20
142	BPF_NEG	= 0x80
143	BPF_OR	= 0x40
144	BPF_RET	= 0x6
145	BPF_RSH	= 0x70
146	BPF_ST	= 0x2
147	BPF_STX	= 0x3
148	BPF_SUB	= 0x10
149	BPF_TAX	= 0x0
150	BPF_TXA	= 0x80
151	BPF_W	= 0x0
152	BPF_X	= 0x8
153	DT_BLK	= 0x6
154	DT_CHR	= 0x2
155	DT_DIR	= 0x4
156	DT_FIFO	= 0x1
157	DT_LNK	= 0xa
158	DT_REG	= 0x8
159	DT_SOCK	= 0xc
160	DT_UNKNOWN	= 0x0
161	DT_WHT	= 0xe
162	EPOLLERR	= 0x8
163	EPOLLET	= -0x80000000
164	EPOLLHUP	= 0x10
165	EPOLLIN	= 0x1
166	EPOLLMSG	= 0x400
167	EPOLLONESHOT	= 0x40000000
168	EPOLLOUT	= 0x4
169	EPOLLPRI	= 0x2
170	EPOLLRDBAND	= 0x80
171	EPOLLRDHUP	= 0x2000
172	EPOLLRDNORM	= 0x40
173	EPOLLWRBAND	= 0x200
174	EPOLLWRNORM	= 0x100
175	EPOLL_CLOEXEC	= 0x80000
176	EPOLL_CTL_ADD	= 0x1
177	EPOLL_CTL_DEL	= 0x2
178	EPOLL_CTL_MOD	= 0x3
179	EPOLL_NONBLOCK	= 0x800
180	ETH_P_1588	= 0x88f7
181	ETH_P_8021Q	= 0x8100
182	ETH_P_802_2	= 0x4
183	ETH_P_802_3	= 0x1
184	ETH_P_AARP	= 0x80f3
185	ETH_P_ALL	= 0x3
186	ETH_P_AOE	= 0x88a2
187	ETH_P_ARCNET	= 0x1a
188	ETH_P_ARP	= 0x806
189	ETH_P_ATALK	= 0x809b

190	ETH_P_ATMFATE	= 0x8884
191	ETH_P_ATMMPOA	= 0x884c
192	ETH_P_AX25	= 0x2
193	ETH_P_BPQ	= 0x8ff
194	ETH_P_CAIF	= 0xf7
195	ETH_P_CAN	= 0xc
196	ETH_P_CONTROL	= 0x16
197	ETH_P_CUST	= 0x6006
198	ETH_P_DDCMP	= 0x6
199	ETH_P_DEC	= 0x6000
200	ETH_P_DIAG	= 0x6005
201	ETH_P_DNA_DL	= 0x6001
202	ETH_P_DNA_RC	= 0x6002
203	ETH_P_DNA_RT	= 0x6003
204	ETH_P_DSA	= 0x1b
205	ETH_P_ECONET	= 0x18
206	ETH_P_EDSA	= 0xdada
207	ETH_P_FCOE	= 0x8906
208	ETH_P_FIP	= 0x8914
209	ETH_P_HDLC	= 0x19
210	ETH_P_IEEE802154	= 0xf6
211	ETH_P_IEEE_PUP	= 0xa00
212	ETH_P_IEEE_PUPAT	= 0xa01
213	ETH_P_IP	= 0x800
214	ETH_P_IPV6	= 0x86dd
215	ETH_P_IPX	= 0x8137
216	ETH_P_IRDA	= 0x17
217	ETH_P_LAT	= 0x6004
218	ETH_P_LINK_CTL	= 0x886c
219	ETH_P_LOCALTALK	= 0x9
220	ETH_P_LOOP	= 0x60
221	ETH_P_MOBITEX	= 0x15
222	ETH_P_MPLS_MC	= 0x8848
223	ETH_P_MPLS_UC	= 0x8847
224	ETH_P_PAE	= 0x888e
225	ETH_P_PAUSE	= 0x8808
226	ETH_P_PHONET	= 0xf5
227	ETH_P_PPPTALK	= 0x10
228	ETH_P_PPP_DISC	= 0x8863
229	ETH_P_PPP_MP	= 0x8
230	ETH_P_PPP_SES	= 0x8864
231	ETH_P_PUP	= 0x200
232	ETH_P_PUPAT	= 0x201
233	ETH_P_RARP	= 0x8035
234	ETH_P_SCA	= 0x6007
235	ETH_P_SLOW	= 0x8809
236	ETH_P_SNAP	= 0x5
237	ETH_P_TEB	= 0x6558
238	ETH_P_TIPC	= 0x88ca
239	ETH_P_TRAILER	= 0x1c

240	ETH_P_TR_802_2	= 0x11
241	ETH_P_WAN_PPP	= 0x7
242	ETH_P_WCCP	= 0x883e
243	ETH_P_X25	= 0x805
244	FD_CLOEXEC	= 0x1
245	FD_SETSIZE	= 0x400
246	F_DUPFD	= 0x0
247	F_DUPFD_CLOEXEC	= 0x406
248	F_EXLCK	= 0x4
249	F_GETFD	= 0x1
250	F_GETFL	= 0x3
251	F_GETLEASE	= 0x401
252	F_GETLK	= 0x5
253	F_GETLK64	= 0x5
254	F_GETOWN	= 0x9
255	F_GETOWN_EX	= 0x10
256	F_GETPIPE_SZ	= 0x408
257	F_GETSIG	= 0xb
258	F_LOCK	= 0x1
259	F_NOTIFY	= 0x402
260	F_OK	= 0x0
261	F_RDLCK	= 0x0
262	F_SETFD	= 0x2
263	F_SETFL	= 0x4
264	F_SETLEASE	= 0x400
265	F_SETLK	= 0x6
266	F_SETLK64	= 0x6
267	F_SETLKW	= 0x7
268	F_SETLKW64	= 0x7
269	F_SETOWN	= 0x8
270	F_SETOWN_EX	= 0xf
271	F_SETPIPE_SZ	= 0x407
272	F_SETSIG	= 0xa
273	F_SHLCK	= 0x8
274	F_TEST	= 0x3
275	F_TLOCK	= 0x2
276	F_ULOCK	= 0x0
277	F_UNLCK	= 0x2
278	F_WRLCK	= 0x1
279	IFA_F_DADFAILED	= 0x8
280	IFA_F_DEPRECATED	= 0x20
281	IFA_F_HOMEADDRESS	= 0x10
282	IFA_F_NODAD	= 0x2
283	IFA_F_OPTIMISTIC	= 0x4
284	IFA_F_PERMANENT	= 0x80
285	IFA_F_SECONDARY	= 0x1
286	IFA_F_TEMPORARY	= 0x1
287	IFA_F_TENTATIVE	= 0x40
288	IFA_MAX	= 0x7

289	IFF_ALLMULTI	= 0x200
290	IFF_AUTOMEDIA	= 0x4000
291	IFF_BROADCAST	= 0x2
292	IFF_DEBUG	= 0x4
293	IFF_DYNAMIC	= 0x8000
294	IFF_LOOPBACK	= 0x8
295	IFF_MASTER	= 0x400
296	IFF_MULTICAST	= 0x1000
297	IFF_NOARP	= 0x80
298	IFF_NOTRAILERS	= 0x20
299	IFF_NO_PI	= 0x1000
300	IFF_ONE_QUEUE	= 0x2000
301	IFF_POINTOPOINT	= 0x10
302	IFF_PORTSEL	= 0x2000
303	IFF_PROMISC	= 0x100
304	IFF_RUNNING	= 0x40
305	IFF_SLAVE	= 0x800
306	IFF_TAP	= 0x2
307	IFF_TUN	= 0x1
308	IFF_TUN_EXCL	= 0x8000
309	IFF_UP	= 0x1
310	IFF_VNET_HDR	= 0x4000
311	IFNAMSIZ	= 0x10
312	IN_ACCESS	= 0x1
313	IN_ALL_EVENTS	= 0xfff
314	IN_ATTRIB	= 0x4
315	IN_CLASSA_HOST	= 0xffffffff
316	IN_CLASSA_MAX	= 0x80
317	IN_CLASSA_NET	= 0xff000000
318	IN_CLASSA_NSHIFT	= 0x18
319	IN_CLASSB_HOST	= 0xffff
320	IN_CLASSB_MAX	= 0x10000
321	IN_CLASSB_NET	= 0xffff0000
322	IN_CLASSB_NSHIFT	= 0x10
323	IN_CLASSC_HOST	= 0xff
324	IN_CLASSC_NET	= 0xfffffff0
325	IN_CLASSC_NSHIFT	= 0x8
326	IN_CLOEXEC	= 0x80000
327	IN_CLOSE	= 0x18
328	IN_CLOSE_NOWRITE	= 0x10
329	IN_CLOSE_WRITE	= 0x8
330	IN_CREATE	= 0x100
331	IN_DELETE	= 0x200
332	IN_DELETE_SELF	= 0x400
333	IN_DONT_FOLLOW	= 0x2000000
334	IN_EXCL_UNLINK	= 0x4000000
335	IN_IGNORED	= 0x8000
336	IN_ISDIR	= 0x40000000
337	IN_LOOPBACKNET	= 0x7f

338	IN_MASK_ADD	= 0x20000000
339	IN_MODIFY	= 0x2
340	IN_MOVE	= 0xc0
341	IN_MOVED_FROM	= 0x40
342	IN_MOVED_TO	= 0x80
343	IN_MOVE_SELF	= 0x800
344	IN_NONBLOCK	= 0x800
345	IN_ONESHOT	= 0x80000000
346	IN_ONLYDIR	= 0x1000000
347	IN_OPEN	= 0x20
348	IN_Q_OVERFLOW	= 0x4000
349	IN_UNMOUNT	= 0x2000
350	IPPROTO_AH	= 0x33
351	IPPROTO_COMP	= 0x6c
352	IPPROTO_DCCP	= 0x21
353	IPPROTO_DSTOPTS	= 0x3c
354	IPPROTO_EGP	= 0x8
355	IPPROTO_ENCAP	= 0x62
356	IPPROTO_ESP	= 0x32
357	IPPROTO_FRAGMENT	= 0x2c
358	IPPROTO_GRE	= 0x2f
359	IPPROTO_HOPOPTS	= 0x0
360	IPPROTO_ICMP	= 0x1
361	IPPROTO_ICMPV6	= 0x3a
362	IPPROTO_IDP	= 0x16
363	IPPROTO_IGMP	= 0x2
364	IPPROTO_IP	= 0x0
365	IPPROTO_IPIP	= 0x4
366	IPPROTO_IPV6	= 0x29
367	IPPROTO_MTP	= 0x5c
368	IPPROTO_NONE	= 0x3b
369	IPPROTO_PIM	= 0x67
370	IPPROTO_PUP	= 0xc
371	IPPROTO_RAW	= 0xff
372	IPPROTO_ROUTING	= 0x2b
373	IPPROTO_RSVP	= 0x2e
374	IPPROTO_SCTP	= 0x84
375	IPPROTO_TCP	= 0x6
376	IPPROTO_TP	= 0x1d
377	IPPROTO_UDP	= 0x11
378	IPPROTO_UDPLITE	= 0x88
379	IPV6_2292DSTOPTS	= 0x4
380	IPV6_2292HOPLIMIT	= 0x8
381	IPV6_2292HOPOPTS	= 0x3
382	IPV6_2292PKTINFO	= 0x2
383	IPV6_2292PKTOPTIONS	= 0x6
384	IPV6_2292RTHDR	= 0x5
385	IPV6_ADDRFORM	= 0x1
386	IPV6_ADD_MEMBERSHIP	= 0x14
387	IPV6_AUTHHDR	= 0xa

388	IPV6_CHECKSUM	= 0x7
389	IPV6_DROP_MEMBERSHIP	= 0x15
390	IPV6_DSTOPTS	= 0x3b
391	IPV6_HOPLIMIT	= 0x34
392	IPV6_HOPOPTS	= 0x36
393	IPV6_IPSEC_POLICY	= 0x22
394	IPV6_JOIN_ANYCAST	= 0x1b
395	IPV6_JOIN_GROUP	= 0x14
396	IPV6_LEAVE_ANYCAST	= 0x1c
397	IPV6_LEAVE_GROUP	= 0x15
398	IPV6_MTU	= 0x18
399	IPV6_MTU_DISCOVER	= 0x17
400	IPV6_MULTICAST_HOPS	= 0x12
401	IPV6_MULTICAST_IF	= 0x11
402	IPV6_MULTICAST_LOOP	= 0x13
403	IPV6_NEXTHOP	= 0x9
404	IPV6_PKTINFO	= 0x32
405	IPV6_PMTUDISC_DO	= 0x2
406	IPV6_PMTUDISC_DONT	= 0x0
407	IPV6_PMTUDISC_PROBE	= 0x3
408	IPV6_PMTUDISC_WANT	= 0x1
409	IPV6_RECVDSOPTS	= 0x3a
410	IPV6_RECVERR	= 0x19
411	IPV6_RECVHOPLIMIT	= 0x33
412	IPV6_RECVHOPOPTS	= 0x35
413	IPV6_RECVPKTINFO	= 0x31
414	IPV6_RECVRTHDR	= 0x38
415	IPV6_RECVTCLASS	= 0x42
416	IPV6_ROUTER_ALERT	= 0x16
417	IPV6_RTHDR	= 0x39
418	IPV6_RTHDRDSOPTS	= 0x37
419	IPV6_RTHDR_LOOSE	= 0x0
420	IPV6_RTHDR_STRICT	= 0x1
421	IPV6_RTHDR_TYPE_0	= 0x0
422	IPV6_RXDSOPTS	= 0x3b
423	IPV6_RXHOPOPTS	= 0x36
424	IPV6_TCLASS	= 0x43
425	IPV6_UNICAST_HOPS	= 0x10
426	IPV6_V6ONLY	= 0x1a
427	IPV6_XFRM_POLICY	= 0x23
428	IP_ADD_MEMBERSHIP	= 0x23
429	IP_ADD_SOURCE_MEMBERSHIP	= 0x27
430	IP_BLOCK_SOURCE	= 0x26
431	IP_DEFAULT_MULTICAST_LOOP	= 0x1
432	IP_DEFAULT_MULTICAST_TTL	= 0x1
433	IP_DF	= 0x4000
434	IP_DROP_MEMBERSHIP	= 0x24
435	IP_DROP_SOURCE_MEMBERSHIP	= 0x28
436	IP_FREEBIND	= 0xf

437	IP_HDRINCL	= 0x3
438	IP_IPSEC_POLICY	= 0x10
439	IP_MAXPACKET	= 0xffff
440	IP_MAX_MEMBERSHIPS	= 0x14
441	IP_MF	= 0x2000
442	IP_MINTTL	= 0x15
443	IP_MSFILTER	= 0x29
444	IP_MSS	= 0x240
445	IP_MTU	= 0xe
446	IP_MTU_DISCOVER	= 0xa
447	IP_MULTICAST_IF	= 0x20
448	IP_MULTICAST_LOOP	= 0x22
449	IP_MULTICAST_TTL	= 0x21
450	IP_OFFMASK	= 0x1fff
451	IP_OPTIONS	= 0x4
452	IP_ORIGDSTADDR	= 0x14
453	IP_PASSEC	= 0x12
454	IP_PKTINFO	= 0x8
455	IP_PKTOPTIONS	= 0x9
456	IP_PMTUDISC	= 0xa
457	IP_PMTUDISC_DO	= 0x2
458	IP_PMTUDISC_DONT	= 0x0
459	IP_PMTUDISC_PROBE	= 0x3
460	IP_PMTUDISC_WANT	= 0x1
461	IP_RECVERR	= 0xb
462	IP_RECVOPTS	= 0x6
463	IP_RECVORIGDSTADDR	= 0x14
464	IP_RECVRETOPTS	= 0x7
465	IP_RECVTOS	= 0xd
466	IP_RECVTTL	= 0xc
467	IP_RETOPTS	= 0x7
468	IP_RF	= 0x8000
469	IP_ROUTER_ALERT	= 0x5
470	IP_TOS	= 0x1
471	IP_TRANSPARENT	= 0x13
472	IP_TTL	= 0x2
473	IP_UNBLOCK_SOURCE	= 0x25
474	IP_XFRM_POLICY	= 0x11
475	LINUX_REBOOT_CMD_CAD_OFF	= 0x0
476	LINUX_REBOOT_CMD_CAD_ON	= 0x89abcdef
477	LINUX_REBOOT_CMD_HALT	= 0xcdef0123
478	LINUX_REBOOT_CMD_KEXEC	= 0x45584543
479	LINUX_REBOOT_CMD_POWER_OFF	= 0x4321fedc
480	LINUX_REBOOT_CMD_RESTART	= 0x1234567
481	LINUX_REBOOT_CMD_RESTART2	= 0xa1b2c3d4
482	LINUX_REBOOT_CMD_SW_SUSPEND	= 0xd000fce2
483	LINUX_REBOOT_MAGIC1	= 0xfce1dead
484	LINUX_REBOOT_MAGIC2	= 0x28121969
485	LOCK_EX	= 0x2

486	LOCK_NB	= 0x4
487	LOCK_SH	= 0x1
488	LOCK_UN	= 0x8
489	MADV_DOFORK	= 0xb
490	MADV_DONTFORK	= 0xa
491	MADV_DONTNEED	= 0x4
492	MADV_HUGEPAGE	= 0xe
493	MADV_HWPOISON	= 0x64
494	MADV_MERGEABLE	= 0xc
495	MADV_NOHUGEPAGE	= 0xf
496	MADV_NORMAL	= 0x0
497	MADV_RANDOM	= 0x1
498	MADV_REMOVE	= 0x9
499	MADV_SEQUENTIAL	= 0x2
500	MADV_UNMERGEABLE	= 0xd
501	MADV_WILLNEED	= 0x3
502	MAP_32BIT	= 0x40
503	MAP_ANON	= 0x20
504	MAP_ANONYMOUS	= 0x20
505	MAP_DENYWRITE	= 0x800
506	MAP_EXECUTABLE	= 0x1000
507	MAP_FILE	= 0x0
508	MAP_FIXED	= 0x10
509	MAP_GROWSDOWN	= 0x100
510	MAP_HUGETLB	= 0x40000
511	MAP_LOCKED	= 0x2000
512	MAP_NONBLOCK	= 0x10000
513	MAP_NORESERVE	= 0x4000
514	MAP_POPULATE	= 0x8000
515	MAP_PRIVATE	= 0x2
516	MAP_SHARED	= 0x1
517	MAP_STACK	= 0x20000
518	MAP_TYPE	= 0xf
519	MCL_CURRENT	= 0x1
520	MCL_FUTURE	= 0x2
521	MNT_DETACH	= 0x2
522	MNT_EXPIRE	= 0x4
523	MNT_FORCE	= 0x1
524	MSG_CMSG_CLOEXEC	= 0x40000000
525	MSG_CONFIRM	= 0x800
526	MSG_CTRUNC	= 0x8
527	MSG_DONTROUTE	= 0x4
528	MSG_DONTWAIT	= 0x40
529	MSG_EOR	= 0x80
530	MSG_ERRQUEUE	= 0x2000
531	MSG_FIN	= 0x200
532	MSG_MORE	= 0x8000
533	MSG_NOSIGNAL	= 0x4000
534	MSG_OOB	= 0x1
535	MSG_PEEK	= 0x2

536	MSG_PROXY	= 0x10
537	MSG_RST	= 0x1000
538	MSG_SYN	= 0x400
539	MSG_TRUNC	= 0x20
540	MSG_TRYHARD	= 0x4
541	MSG_WAITALL	= 0x100
542	MSG_WAITFORONE	= 0x10000
543	MS_ACTIVE	= 0x40000000
544	MS_ASYNC	= 0x1
545	MS_BIND	= 0x1000
546	MS_DIRSYNC	= 0x80
547	MS_INVALIDATE	= 0x2
548	MS_I_VERSION	= 0x800000
549	MS_KERNMOUNT	= 0x400000
550	MS_MANDLOCK	= 0x40
551	MS_MGC_MSK	= 0xffff0000
552	MS_MGC_VAL	= 0xc0ed0000
553	MS_MOVE	= 0x2000
554	MS_NOATIME	= 0x400
555	MS_NODEV	= 0x4
556	MS_NODIRATIME	= 0x800
557	MS_NOEXEC	= 0x8
558	MS_NOSUID	= 0x2
559	MS_NOUSER	= -0x80000000
560	MS_POSIXACL	= 0x10000
561	MS_PRIVATE	= 0x40000
562	MS_RDONLY	= 0x1
563	MS_REC	= 0x4000
564	MS_RELATIME	= 0x200000
565	MS_REMOUNT	= 0x20
566	MS_RMT_MASK	= 0x800051
567	MS_SHARED	= 0x100000
568	MS_SILENT	= 0x8000
569	MS_SLAVE	= 0x80000
570	MS_STRICTATIME	= 0x1000000
571	MS_SYNC	= 0x4
572	MS_SYNCHRONOUS	= 0x10
573	MS_UNBINDABLE	= 0x20000
574	NAME_MAX	= 0xff
575	NETLINK_ADD_MEMBERSHIP	= 0x1
576	NETLINK_AUDIT	= 0x9
577	NETLINK_BROADCAST_ERROR	= 0x4
578	NETLINK_CONNECTOR	= 0xb
579	NETLINK_DNRTMSG	= 0xe
580	NETLINK_DROP_MEMBERSHIP	= 0x2
581	NETLINK_ECRYPTFS	= 0x13
582	NETLINK_FIB_LOOKUP	= 0xa
583	NETLINK_FIREWALL	= 0x3
584	NETLINK_GENERIC	= 0x10

585	NETLINK_INET_DIAG	= 0x4
586	NETLINK_IP6_FW	= 0xd
587	NETLINK_ISCSI	= 0x8
588	NETLINK_KOBJECT_UEVENT	= 0xf
589	NETLINK_NETFILTER	= 0xc
590	NETLINK_NFLOG	= 0x5
591	NETLINK_NO_ENOBUFS	= 0x5
592	NETLINK_PKTINFO	= 0x3
593	NETLINK_ROUTE	= 0x0
594	NETLINK_SCSITRANSPORT	= 0x12
595	NETLINK_SELINUX	= 0x7
596	NETLINK_UNUSED	= 0x1
597	NETLINK_USERSOCK	= 0x2
598	NETLINK_XFRM	= 0x6
599	NLA_ALIGNTO	= 0x4
600	NLA_F_NESTED	= 0x8000
601	NLA_F_NET_BYTEORDER	= 0x4000
602	NLA_HDRLEN	= 0x4
603	NLMSG_ALIGNTO	= 0x4
604	NLMSG_DONE	= 0x3
605	NLMSG_ERROR	= 0x2
606	NLMSG_HDRLEN	= 0x10
607	NLMSG_MIN_TYPE	= 0x10
608	NLMSG_NOOP	= 0x1
609	NLMSG_OVERRUN	= 0x4
610	NLM_F_ACK	= 0x4
611	NLM_F_APPEND	= 0x800
612	NLM_F_ATOMIC	= 0x400
613	NLM_F_CREATE	= 0x400
614	NLM_F_DUMP	= 0x300
615	NLM_F_ECHO	= 0x8
616	NLM_F_EXCL	= 0x200
617	NLM_F_MATCH	= 0x200
618	NLM_F_MULTI	= 0x2
619	NLM_F_REPLACE	= 0x100
620	NLM_F_REQUEST	= 0x1
621	NLM_F_ROOT	= 0x100
622	O_ACCMODE	= 0x3
623	O_APPEND	= 0x400
624	O_ASYNC	= 0x2000
625	O_CLOEXEC	= 0x80000
626	O_CREAT	= 0x40
627	O_DIRECT	= 0x4000
628	O_DIRECTORY	= 0x10000
629	O_DSYNC	= 0x1000
630	O_EXCL	= 0x80
631	O_FSYNC	= 0x101000
632	O_LARGEFILE	= 0x0
633	O_NDELAY	= 0x800

634	O_NOATIME	= 0x40000
635	O_NOCTTY	= 0x100
636	O_NOFOLLOW	= 0x20000
637	O_NONBLOCK	= 0x800
638	O_RDONLY	= 0x0
639	O_RDWR	= 0x2
640	O_RSYNC	= 0x101000
641	O_SYNC	= 0x101000
642	O_TRUNC	= 0x200
643	O_WRONLY	= 0x1
644	PACKET_ADD_MEMBERSHIP	= 0x1
645	PACKET_BROADCAST	= 0x1
646	PACKET_DROP_MEMBERSHIP	= 0x2
647	PACKET_FASTROUTE	= 0x6
648	PACKET_HOST	= 0x0
649	PACKET_LOOPBACK	= 0x5
650	PACKET_MR_ALLMULTI	= 0x2
651	PACKET_MR_MULTICAST	= 0x0
652	PACKET_MR_PROMISC	= 0x1
653	PACKET_MULTICAST	= 0x2
654	PACKET_OTHERHOST	= 0x3
655	PACKET_OUTGOING	= 0x4
656	PACKET_RECV_OUTPUT	= 0x3
657	PACKET_RX_RING	= 0x5
658	PACKET_STATISTICS	= 0x6
659	PROT_EXEC	= 0x4
660	PROT_GROWSDOWN	= 0x1000000
661	PROT_GROWSUP	= 0x2000000
662	PROT_NONE	= 0x0
663	PROT_READ	= 0x1
664	PROT_WRITE	= 0x2
665	PR_CAPBSET_DROP	= 0x18
666	PR_CAPBSET_READ	= 0x17
667	PR_ENDIAN_BIG	= 0x0
668	PR_ENDIAN_LITTLE	= 0x1
669	PR_ENDIAN_PPC_LITTLE	= 0x2
670	PR_FPEMU_NOPRINT	= 0x1
671	PR_FPEMU_SIGFPE	= 0x2
672	PR_FP_EXC_ASYNC	= 0x2
673	PR_FP_EXC_DISABLED	= 0x0
674	PR_FP_EXC_DIV	= 0x10000
675	PR_FP_EXC_INV	= 0x100000
676	PR_FP_EXC_NONRECOV	= 0x1
677	PR_FP_EXC_OVF	= 0x20000
678	PR_FP_EXC_PRECISE	= 0x3
679	PR_FP_EXC_RES	= 0x80000
680	PR_FP_EXC_SW_ENABLE	= 0x80
681	PR_FP_EXC_UND	= 0x40000
682	PR_GET_DUMPABLE	= 0x3
683	PR_GET_ENDIAN	= 0x13

684	PR_GET_FPEMU	= 0x9
685	PR_GET_FPEXC	= 0xb
686	PR_GET_KEEPCAPS	= 0x7
687	PR_GET_NAME	= 0x10
688	PR_GET_PDEATHSIG	= 0x2
689	PR_GET_SECCOMP	= 0x15
690	PR_GET_SECUREBITS	= 0x1b
691	PR_GET_TIMERSLACK	= 0x1e
692	PR_GET_TIMING	= 0xd
693	PR_GET_TSC	= 0x19
694	PR_GET_UNALIGN	= 0x5
695	PR_MCE_KILL	= 0x21
696	PR_MCE_KILL_CLEAR	= 0x0
697	PR_MCE_KILL_DEFAULT	= 0x2
698	PR_MCE_KILL_EARLY	= 0x1
699	PR_MCE_KILL_GET	= 0x22
700	PR_MCE_KILL_LATE	= 0x0
701	PR_MCE_KILL_SET	= 0x1
702	PR_SET_DUMPABLE	= 0x4
703	PR_SET_ENDIAN	= 0x14
704	PR_SET_FPEMU	= 0xa
705	PR_SET_FPEXC	= 0xc
706	PR_SET_KEEPCAPS	= 0x8
707	PR_SET_NAME	= 0xf
708	PR_SET_PDEATHSIG	= 0x1
709	PR_SET_PTRACER	= 0x59616d61
710	PR_SET_SECCOMP	= 0x16
711	PR_SET_SECUREBITS	= 0x1c
712	PR_SET_TIMERSLACK	= 0x1d
713	PR_SET_TIMING	= 0xe
714	PR_SET_TSC	= 0x1a
715	PR_SET_UNALIGN	= 0x6
716	PR_TASK_PERF_EVENTS_DISABLE	= 0x1f
717	PR_TASK_PERF_EVENTS_ENABLE	= 0x20
718	PR_TIMING_STATISTICAL	= 0x0
719	PR_TIMING_TIMESTAMP	= 0x1
720	PR_TSC_ENABLE	= 0x1
721	PR_TSC_SIGSEGV	= 0x2
722	PR_UNALIGN_NOPRINT	= 0x1
723	PR_UNALIGN_SIGBUS	= 0x2
724	PTRACE_ARCH_PRCTL	= 0x1e
725	PTRACE_ATTACH	= 0x10
726	PTRACE_CONT	= 0x7
727	PTRACE_DETACH	= 0x11
728	PTRACE_EVENT_CLONE	= 0x3
729	PTRACE_EVENT_EXEC	= 0x4
730	PTRACE_EVENT_EXIT	= 0x6
731	PTRACE_EVENT_FORK	= 0x1
732	PTRACE_EVENT_VFORK	= 0x2

733	PTRACE_EVENT_VFORK_DONE	= 0x5
734	PTRACE_GETEVENTMSG	= 0x4201
735	PTRACE_GETFPREGS	= 0xe
736	PTRACE_GETFPXREGS	= 0x12
737	PTRACE_GETREGS	= 0xc
738	PTRACE_GETREGSET	= 0x4204
739	PTRACE_GETSIGINFO	= 0x4202
740	PTRACE_GET_THREAD_AREA	= 0x19
741	PTRACE_KILL	= 0x8
742	PTRACE_OLDSETOPTIONS	= 0x15
743	PTRACE_O_MASK	= 0x7f
744	PTRACE_O_TRACECLONE	= 0x8
745	PTRACE_O_TRACEEXEC	= 0x10
746	PTRACE_O_TRACEEXIT	= 0x40
747	PTRACE_O_TRACEFORK	= 0x2
748	PTRACE_O_TRACESYSGOOD	= 0x1
749	PTRACE_O_TRACEVFORK	= 0x4
750	PTRACE_O_TRACEVFORKDONE	= 0x20
751	PTRACE_PEEKDATA	= 0x2
752	PTRACE_PEEKTEXT	= 0x1
753	PTRACE_PEEKUSR	= 0x3
754	PTRACE_POKEDATA	= 0x5
755	PTRACE_POKETEXT	= 0x4
756	PTRACE_POKEUSR	= 0x6
757	PTRACE_SETFPREGS	= 0xf
758	PTRACE_SETFPXREGS	= 0x13
759	PTRACE_SETOPTIONS	= 0x4200
760	PTRACE_SETREGS	= 0xd
761	PTRACE_SETREGSET	= 0x4205
762	PTRACE_SETSIGINFO	= 0x4203
763	PTRACE_SET_THREAD_AREA	= 0x1a
764	PTRACE_SINGLEBLOCK	= 0x21
765	PTRACE_SINGLESTEP	= 0x9
766	PTRACE_SYSCALL	= 0x18
767	PTRACE_SYSEMU	= 0x1f
768	PTRACE_SYSEMU_SINGLESTEP	= 0x20
769	PTRACE_TRACEME	= 0x0
770	RLIMIT_AS	= 0x9
771	RLIMIT_CORE	= 0x4
772	RLIMIT_CPU	= 0x0
773	RLIMIT_DATA	= 0x2
774	RLIMIT_FSIZE	= 0x1
775	RLIMIT_NOFILE	= 0x7
776	RLIMIT_STACK	= 0x3
777	RLIM_INFINITY	= -0x1
778	RTAX_ADVMSS	= 0x8
779	RTAX_CWND	= 0x7
780	RTAX_FEATURES	= 0xc
781	RTAX_FEATURE_ALLFRAG	= 0x8

782	RTAX_FEATURE_ECN	= 0x1
783	RTAX_FEATURE_SACK	= 0x2
784	RTAX_FEATURE_TIMESTAMP	= 0x4
785	RTAX_HOPLIMIT	= 0xa
786	RTAX_INITCWND	= 0xb
787	RTAX_INITRWND	= 0xe
788	RTAX_LOCK	= 0x1
789	RTAX_MAX	= 0xe
790	RTAX_MTU	= 0x2
791	RTAX_REORDERING	= 0x9
792	RTAX_RTO_MIN	= 0xd
793	RTAX_RTT	= 0x4
794	RTAX_RTTVAR	= 0x5
795	RTAX_SSTHRESH	= 0x6
796	RTAX_UNSPEC	= 0x0
797	RTAX_WINDOW	= 0x3
798	RTA_ALIGNTO	= 0x4
799	RTA_MAX	= 0x10
800	RTCF_DIRECTSRC	= 0x4000000
801	RTCF_DOREDIRECT	= 0x1000000
802	RTCF_LOG	= 0x2000000
803	RTCF_MASQ	= 0x400000
804	RTCF_NAT	= 0x800000
805	RTCF_VALVE	= 0x200000
806	RTF_ADDRCLASSMASK	= 0xf8000000
807	RTF_ADDRCONF	= 0x40000
808	RTF_ALLONLINK	= 0x20000
809	RTF_BROADCAST	= 0x10000000
810	RTF_CACHE	= 0x1000000
811	RTF_DEFAULT	= 0x10000
812	RTF_DYNAMIC	= 0x10
813	RTF_FLOW	= 0x2000000
814	RTF_GATEWAY	= 0x2
815	RTF_HOST	= 0x4
816	RTF_INTERFACE	= 0x40000000
817	RTF_IRTT	= 0x100
818	RTF_LINKRT	= 0x100000
819	RTF_LOCAL	= 0x80000000
820	RTF_MODIFIED	= 0x20
821	RTF_MSS	= 0x40
822	RTF_MTU	= 0x40
823	RTF_MULTICAST	= 0x20000000
824	RTF_NAT	= 0x8000000
825	RTF_NOFORWARD	= 0x1000
826	RTF_NONEXTHOP	= 0x200000
827	RTF_NOPMTUDISC	= 0x4000
828	RTF_POLICY	= 0x4000000
829	RTF_REINSTATE	= 0x8
830	RTF_REJECT	= 0x200
831	RTF_STATIC	= 0x400

832	RTF_THROW	= 0x2000
833	RTF_UP	= 0x1
834	RTF_WINDOW	= 0x80
835	RTF_XRESOLVE	= 0x800
836	RTM_BASE	= 0x10
837	RTM_DELACTION	= 0x31
838	RTM_DELADDR	= 0x15
839	RTM_DELADDRLABEL	= 0x49
840	RTM_DELLINK	= 0x11
841	RTM_DELNEIGH	= 0x1d
842	RTM_DELQDISC	= 0x25
843	RTM_DELROUTE	= 0x19
844	RTM_DELRULE	= 0x21
845	RTM_DELTCLASS	= 0x29
846	RTM_DELTFILTER	= 0x2d
847	RTM_F_CLONED	= 0x200
848	RTM_F_EQUALIZE	= 0x400
849	RTM_F_NOTIFY	= 0x100
850	RTM_F_PREFIX	= 0x800
851	RTM_GETACTION	= 0x32
852	RTM_GETADDR	= 0x16
853	RTM_GETADDRLABEL	= 0x4a
854	RTM_GETANYCAST	= 0x3e
855	RTM_GETDCB	= 0x4e
856	RTM_GETLINK	= 0x12
857	RTM_GETMULTICAST	= 0x3a
858	RTM_GETNEIGH	= 0x1e
859	RTM_GETNEIGHTBL	= 0x42
860	RTM_GETQDISC	= 0x26
861	RTM_GETROUTE	= 0x1a
862	RTM_GETRULE	= 0x22
863	RTM_GETTCLASS	= 0x2a
864	RTM_GETTFILTER	= 0x2e
865	RTM_MAX	= 0x4f
866	RTM_NEWACTION	= 0x30
867	RTM_NEWADDR	= 0x14
868	RTM_NEWADDRLABEL	= 0x48
869	RTM_NEWLINK	= 0x10
870	RTM_NEWNDUSEROPT	= 0x44
871	RTM_NEWNEIGH	= 0x1c
872	RTM_NEWNEIGHTBL	= 0x40
873	RTM_NEWPREFIX	= 0x34
874	RTM_NEWQDISC	= 0x24
875	RTM_NEWROUTE	= 0x18
876	RTM_NEWRULE	= 0x20
877	RTM_NEWTCLASS	= 0x28
878	RTM_NEWTFILTER	= 0x2c
879	RTM_NR_FAMILIES	= 0x10
880	RTM_NR_MSGTYPES	= 0x40

881	RTM_SETDCB	= 0x4f
882	RTM_SETLINK	= 0x13
883	RTM_SETNEIGHTBL	= 0x43
884	RTNH_ALIGNTO	= 0x4
885	RTNH_F_DEAD	= 0x1
886	RTNH_F_ONLINK	= 0x4
887	RTNH_F_PERVASIVE	= 0x2
888	RTN_MAX	= 0xb
889	RTPROT_BIRD	= 0xc
890	RTPROT_BOOT	= 0x3
891	RTPROT_DHCP	= 0x10
892	RTPROT_DNRROUTED	= 0xd
893	RTPROT_GATED	= 0x8
894	RTPROT_KERNEL	= 0x2
895	RTPROT_MRT	= 0xa
896	RTPROT_NTK	= 0xf
897	RTPROT_RA	= 0x9
898	RTPROT_REDIRECT	= 0x1
899	RTPROT_STATIC	= 0x4
900	RTPROT_UNSPEC	= 0x0
901	RTPROT_XORP	= 0xe
902	RTPROT_ZEBRA	= 0xb
903	RT_CLASS_DEFAULT	= 0xfd
904	RT_CLASS_LOCAL	= 0xff
905	RT_CLASS_MAIN	= 0xfe
906	RT_CLASS_MAX	= 0xff
907	RT_CLASS_UNSPEC	= 0x0
908	RUSAGE_CHILDREN	= -0x1
909	RUSAGE_SELF	= 0x0
910	RUSAGE_THREAD	= 0x1
911	SCM_CREDENTIALS	= 0x2
912	SCM_RIGHTS	= 0x1
913	SCM_TIMESTAMP	= 0x1d
914	SCM_TIMESTAMPING	= 0x25
915	SCM_TIMESTAMPNS	= 0x23
916	SHUT_RD	= 0x0
917	SHUT_RDWR	= 0x2
918	SHUT_WR	= 0x1
919	SIOCADDLCI	= 0x8980
920	SIOCADDMULTI	= 0x8931
921	SIOCADDRT	= 0x890b
922	SIOCATMARK	= 0x8905
923	SIOCDDARP	= 0x8953
924	SIOCDELDLCI	= 0x8981
925	SIOCDELMULTI	= 0x8932
926	SIOCDELRT	= 0x890c
927	SIOCDEVPRIVATE	= 0x89f0
928	SIOCDFADDR	= 0x8936
929	SIOCRRARP	= 0x8960

930	SIOCGARP	= 0x8954
931	SIOCGIFADDR	= 0x8915
932	SIOCGIFBR	= 0x8940
933	SIOCGIFBRDADDR	= 0x8919
934	SIOCGIFCONF	= 0x8912
935	SIOCGIFCOUNT	= 0x8938
936	SIOCGIFDSTADDR	= 0x8917
937	SIOCGIFENCAP	= 0x8925
938	SIOCGIFFLAGS	= 0x8913
939	SIOCGIFHWADDR	= 0x8927
940	SIOCGIFINDEX	= 0x8933
941	SIOCGIFMAP	= 0x8970
942	SIOCGIFMEM	= 0x891f
943	SIOCGIFMETRIC	= 0x891d
944	SIOCGIFMTU	= 0x8921
945	SIOCGIFNAME	= 0x8910
946	SIOCGIFNETMASK	= 0x891b
947	SIOCGIFPFLAGS	= 0x8935
948	SIOCGIFSLAVE	= 0x8929
949	SIOCGIFTXQLEN	= 0x8942
950	SIOCGPGRP	= 0x8904
951	SIOCGRARP	= 0x8961
952	SIOCGSTAMP	= 0x8906
953	SIOCGSTAMPNS	= 0x8907
954	SIOCPROTOPRIVATE	= 0x89e0
955	SIOCRTMSG	= 0x890d
956	SIOCSARP	= 0x8955
957	SIOCSIFADDR	= 0x8916
958	SIOCSIFBR	= 0x8941
959	SIOCSIFBRDADDR	= 0x891a
960	SIOCSIFDSTADDR	= 0x8918
961	SIOCSIFENCAP	= 0x8926
962	SIOCSIFFLAGS	= 0x8914
963	SIOCSIFHWADDR	= 0x8924
964	SIOCSIFHWBROADCAST	= 0x8937
965	SIOCSIFLINK	= 0x8911
966	SIOCSIFMAP	= 0x8971
967	SIOCSIFMEM	= 0x8920
968	SIOCSIFMETRIC	= 0x891e
969	SIOCSIFMTU	= 0x8922
970	SIOCSIFNAME	= 0x8923
971	SIOCSIFNETMASK	= 0x891c
972	SIOCSIFPFLAGS	= 0x8934
973	SIOCSIFSLAVE	= 0x8930
974	SIOCSIFTXQLEN	= 0x8943
975	SIOCSPPGRP	= 0x8902
976	SIOCSRARP	= 0x8962
977	SOCK_CLOEXEC	= 0x80000
978	SOCK_DCCP	= 0x6
979	SOCK_DGRAM	= 0x2

980	SOCK_NONBLOCK	= 0x800
981	SOCK_PACKET	= 0xa
982	SOCK_RAW	= 0x3
983	SOCK_RDM	= 0x4
984	SOCK_SEQPACKET	= 0x5
985	SOCK_STREAM	= 0x1
986	SOL_AAL	= 0x109
987	SOL_ATM	= 0x108
988	SOL_DECNET	= 0x105
989	SOL_ICMPV6	= 0x3a
990	SOL_IP	= 0x0
991	SOL_IPV6	= 0x29
992	SOL_IRDA	= 0x10a
993	SOL_PACKET	= 0x107
994	SOL_RAW	= 0xff
995	SOL_SOCKET	= 0x1
996	SOL_TCP	= 0x6
997	SOL_X25	= 0x106
998	SOMAXCONN	= 0x80
999	SO_ACCEPTCONN	= 0x1e
1000	SO_ATTACH_FILTER	= 0x1a
1001	SO_BINDTODEVICE	= 0x19
1002	SO_BROADCAST	= 0x6
1003	SO_BSDCOMPAT	= 0xe
1004	SO_DEBUG	= 0x1
1005	SO_DETACH_FILTER	= 0x1b
1006	SO_DOMAIN	= 0x27
1007	SO_DONTROUTE	= 0x5
1008	SO_ERROR	= 0x4
1009	SO_KEEPALIVE	= 0x9
1010	SO_LINGER	= 0xd
1011	SO_MARK	= 0x24
1012	SO_NO_CHECK	= 0xb
1013	SO_OOBINLINE	= 0xa
1014	SO_PASSCRED	= 0x10
1015	SO_PASSEC	= 0x22
1016	SO_PEERCREC	= 0x11
1017	SO_PEERNAME	= 0x1c
1018	SO_PEERSEC	= 0x1f
1019	SO_PRIORITY	= 0xc
1020	SO_PROTOCOL	= 0x26
1021	SO_RCVBUF	= 0x8
1022	SO_RCVBUFFORCE	= 0x21
1023	SO_RCVLOWAT	= 0x12
1024	SO_RCVTIMEO	= 0x14
1025	SO_REUSEADDR	= 0x2
1026	SO_RXQ_OVFL	= 0x28
1027	SO_SECURITY_AUTHENTICATION	= 0x16
1028	SO_SECURITY_ENCRYPTION_NETWORK	= 0x18

1029	SO_SECURITY_ENCRYPTION_TRANSPORT	= 0x17
1030	SO_SNDBUF	= 0x7
1031	SO_SNDBUFFORCE	= 0x20
1032	SO_SNDLOWAT	= 0x13
1033	SO_SNDTIMEO	= 0x15
1034	SO_TIMESTAMP	= 0x1d
1035	SO_TIMESTAMPING	= 0x25
1036	SO_TIMESTAMPNS	= 0x23
1037	SO_TYPE	= 0x3
1038	S_BLKSIZE	= 0x200
1039	S_IEXEC	= 0x40
1040	S_IFBLK	= 0x6000
1041	S_IFCHR	= 0x2000
1042	S_IFDIR	= 0x4000
1043	S_IFIFO	= 0x1000
1044	S_IFLNK	= 0xa000
1045	S_IFMT	= 0xf000
1046	S_IFREG	= 0x8000
1047	S_IFSOCK	= 0xc000
1048	S_IREAD	= 0x100
1049	S_IRGRP	= 0x20
1050	S_IROTH	= 0x4
1051	S_IRUSR	= 0x100
1052	S_IRWXG	= 0x38
1053	S_IRWXO	= 0x7
1054	S_IRWXU	= 0x1c0
1055	S_ISGID	= 0x400
1056	S_ISUID	= 0x800
1057	S_ISVTX	= 0x200
1058	S_IWGRP	= 0x10
1059	S_IWOTH	= 0x2
1060	S_IWRITE	= 0x80
1061	S_IWUSR	= 0x80
1062	S_IXGRP	= 0x8
1063	S_IXOTH	= 0x1
1064	S_IXUSR	= 0x40
1065	TCP_CONGESTION	= 0xd
1066	TCP_CORK	= 0x3
1067	TCP_DEFER_ACCEPT	= 0x9
1068	TCP_INFO	= 0xb
1069	TCP_KEEPCNT	= 0x6
1070	TCP_KEEPIDLE	= 0x4
1071	TCP_KEEPINTVL	= 0x5
1072	TCP_LINGER2	= 0x8
1073	TCP_MAXSEG	= 0x2
1074	TCP_MAXWIN	= 0xffff
1075	TCP_MAX_WINSHIFT	= 0xe
1076	TCP_MD5SIG	= 0xe
1077	TCP_MD5SIG_MAXKEYLEN	= 0x50

1078	TCP_MSS	= 0x200
1079	TCP_NODELAY	= 0x1
1080	TCP_QUICKACK	= 0xc
1081	TCP_SYNCNT	= 0x7
1082	TCP_WINDOW_CLAMP	= 0xa
1083	TIOCCBRK	= 0x5428
1084	TIOCCONS	= 0x541d
1085	TIOCEXCL	= 0x540c
1086	TIOCGDEV	= 0x80045432
1087	TIOCGETD	= 0x5424
1088	TIOCGICOUNT	= 0x545d
1089	TIOCGLOCKTRMIOS	= 0x5456
1090	TIOCGPGRP	= 0x540f
1091	TIOCGPTN	= 0x80045430
1092	TIOCGRS485	= 0x542e
1093	TIOCGSERIAL	= 0x541e
1094	TIOCGSID	= 0x5429
1095	TIOCGSOFTCAR	= 0x5419
1096	TIOCGWINSZ	= 0x5413
1097	TIOCINQ	= 0x541b
1098	TIOCLINUX	= 0x541c
1099	TIOCMBIC	= 0x5417
1100	TIOCMBIS	= 0x5416
1101	TIOCMGET	= 0x5415
1102	TIOCMWAIT	= 0x545c
1103	TIOCMSET	= 0x5418
1104	TIOCM_CAR	= 0x40
1105	TIOCM_CD	= 0x40
1106	TIOCM_CTS	= 0x20
1107	TIOCM_DSR	= 0x100
1108	TIOCM_DTR	= 0x2
1109	TIOCM_LE	= 0x1
1110	TIOCM_RI	= 0x80
1111	TIOCM_RNG	= 0x80
1112	TIOCM_RTS	= 0x4
1113	TIOCM_SR	= 0x10
1114	TIOCM_ST	= 0x8
1115	TIOCNOTTY	= 0x5422
1116	TIOCNXCL	= 0x540d
1117	TIOCOUTQ	= 0x5411
1118	TIOCPKT	= 0x5420
1119	TIOCPKT_DATA	= 0x0
1120	TIOCPKT_DOSTOP	= 0x20
1121	TIOCPKT_FLUSHREAD	= 0x1
1122	TIOCPKT_FLUSHWRITE	= 0x2
1123	TIOCPKT_IOCTL	= 0x40
1124	TIOCPKT_NOSTOP	= 0x10
1125	TIOCPKT_START	= 0x8
1126	TIOCPKT_STOP	= 0x4
1127	TIOCSBRK	= 0x5427

1128	TIOCSCTTY	= 0x540e
1129	TIOCSERCONFIG	= 0x5453
1130	TIOCSERGETLSR	= 0x5459
1131	TIOCSERGETMULTI	= 0x545a
1132	TIOCSERGSTRUCT	= 0x5458
1133	TIOCSERGWILD	= 0x5454
1134	TIOCSERSETMULTI	= 0x545b
1135	TIOCSERSWILD	= 0x5455
1136	TIOCSER_TEMT	= 0x1
1137	TIOCSETD	= 0x5423
1138	TIOCSIG	= 0x40045436
1139	TIOCSLCKTRMIO	= 0x5457
1140	TIOCSPGRP	= 0x5410
1141	TIOCSPTLCK	= 0x40045431
1142	TIOCSRS485	= 0x542f
1143	TIOCSSERIAL	= 0x541f
1144	TIOCSSOFTCAR	= 0x541a
1145	TIOCSTI	= 0x5412
1146	TIOCSWINSZ	= 0x5414
1147	TUNATTACHFILTER	= 0x401054d5
1148	TUNDETACHFILTER	= 0x401054d6
1149	TUNGETFEATURES	= 0x800454cf
1150	TUNGETIFF	= 0x800454d2
1151	TUNGETSNDBUF	= 0x800454d3
1152	TUNGETVNETHDRSZ	= 0x800454d7
1153	TUNSETDEBUG	= 0x400454c9
1154	TUNSETGROUP	= 0x400454ce
1155	TUNSETIFF	= 0x400454ca
1156	TUNSETLINK	= 0x400454cd
1157	TUNSETNOCSUM	= 0x400454c8
1158	TUNSETOFFLOAD	= 0x400454d0
1159	TUNSETOWNER	= 0x400454cc
1160	TUNSETPERSIST	= 0x400454cb
1161	TUNSETSNDBUF	= 0x400454d4
1162	TUNSETTXFILTER	= 0x400454d1
1163	TUNSETVNETHDRSZ	= 0x400454d8
1164	WALL	= 0x40000000
1165	WCLONE	= 0x80000000
1166	WCONTINUED	= 0x8
1167	WEXITED	= 0x4
1168	WNOHANG	= 0x1
1169	WNOTHREAD	= 0x20000000
1170	WNOWAIT	= 0x1000000
1171	WORDSIZE	= 0x40
1172	WSTOPPED	= 0x2
1173	WUNTRACED	= 0x2
1174)	
1175		
1176	// Errors	

```
1177 const (
1178     E2BIG           = Errno(0x7)
1179     EACCES          = Errno(0xd)
1180     EADDRINUSE     = Errno(0x62)
1181     EADDRNOTAVAIL = Errno(0x63)
1182     EADV           = Errno(0x44)
1183     EAFNOSUPPORT  = Errno(0x61)
1184     EAGAIN        = Errno(0xb)
1185     EALREADY     = Errno(0x72)
1186     EBADE        = Errno(0x34)
1187     EBADF        = Errno(0x9)
1188     EBADFD      = Errno(0x4d)
1189     EBADMSG     = Errno(0x4a)
1190     EBADR       = Errno(0x35)
1191     EBADRQC    = Errno(0x38)
1192     EBADSLT    = Errno(0x39)
1193     EBFONT     = Errno(0x3b)
1194     EBUSY      = Errno(0x10)
1195     ECANCELED  = Errno(0x7d)
1196     ECHILD     = Errno(0xa)
1197     ECHRNG    = Errno(0x2c)
1198     ECOMM     = Errno(0x46)
1199     ECONNABORTED = Errno(0x67)
1200     ECONNREFUSED = Errno(0x6f)
1201     ECONNRESET = Errno(0x68)
1202     EDEADLK   = Errno(0x23)
1203     EDEADLOCK = Errno(0x23)
1204     EDESTADDRREQ = Errno(0x59)
1205     EDOM      = Errno(0x21)
1206     EDOTDOT   = Errno(0x49)
1207     EDQUOT    = Errno(0x7a)
1208     EEXIST    = Errno(0x11)
1209     EFAULT    = Errno(0xe)
1210     EFBIG     = Errno(0x1b)
1211     EHOSTDOWN = Errno(0x70)
1212     EHOSTUNREACH = Errno(0x71)
1213     EIDRM     = Errno(0x2b)
1214     EILSEQ    = Errno(0x54)
1215     EINPROGRESS = Errno(0x73)
1216     EINTR     = Errno(0x4)
1217     EINVAL    = Errno(0x16)
1218     EIO       = Errno(0x5)
1219     EISCONN   = Errno(0x6a)
1220     EISDIR    = Errno(0x15)
1221     EISNAM    = Errno(0x78)
1222     EKEYEXPIRED = Errno(0x7f)
1223     EKEYREJECTED = Errno(0x81)
1224     EKEYREVOKED = Errno(0x80)
1225     EL2HLT    = Errno(0x33)
```

1226	EL2NSYNC	= Errno(0x2d)
1227	EL3HLT	= Errno(0x2e)
1228	EL3RST	= Errno(0x2f)
1229	ELIBACC	= Errno(0x4f)
1230	ELIBBAD	= Errno(0x50)
1231	ELIBEXEC	= Errno(0x53)
1232	ELIBMAX	= Errno(0x52)
1233	ELIBSCN	= Errno(0x51)
1234	ELNRNG	= Errno(0x30)
1235	ELOOP	= Errno(0x28)
1236	EMEDIUMTYPE	= Errno(0x7c)
1237	EMFILE	= Errno(0x18)
1238	EMLINK	= Errno(0x1f)
1239	EMSGSIZE	= Errno(0x5a)
1240	EMULTIHOP	= Errno(0x48)
1241	ENAMETOOLONG	= Errno(0x24)
1242	ENAVAIL	= Errno(0x77)
1243	ENETDOWN	= Errno(0x64)
1244	ENETRESET	= Errno(0x66)
1245	ENETUNREACH	= Errno(0x65)
1246	ENFILE	= Errno(0x17)
1247	ENOANO	= Errno(0x37)
1248	ENOBUFS	= Errno(0x69)
1249	ENOCSI	= Errno(0x32)
1250	ENODATA	= Errno(0x3d)
1251	ENODEV	= Errno(0x13)
1252	ENOENT	= Errno(0x2)
1253	ENOEXEC	= Errno(0x8)
1254	ENOKEY	= Errno(0x7e)
1255	ENOLCK	= Errno(0x25)
1256	ENOLINK	= Errno(0x43)
1257	ENOMEDIUM	= Errno(0x7b)
1258	ENOMEM	= Errno(0xc)
1259	ENOMSG	= Errno(0x2a)
1260	ENONET	= Errno(0x40)
1261	ENOPKG	= Errno(0x41)
1262	ENOPROTOPT	= Errno(0x5c)
1263	ENOSPC	= Errno(0x1c)
1264	ENOSR	= Errno(0x3f)
1265	ENOSTR	= Errno(0x3c)
1266	ENOSYS	= Errno(0x26)
1267	ENOTBLK	= Errno(0xf)
1268	ENOTCONN	= Errno(0x6b)
1269	ENOTDIR	= Errno(0x14)
1270	ENOTEMPTY	= Errno(0x27)
1271	ENOTNAM	= Errno(0x76)
1272	ENOTRECOVERABLE	= Errno(0x83)
1273	ENOTSOCK	= Errno(0x58)
1274	ENOTSUP	= Errno(0x5f)
1275	ENOTTY	= Errno(0x19)

```

1276      ENOTUNIQ      = Errno(0x4c)
1277      ENXIO          = Errno(0x6)
1278      EOPNOTSUPP     = Errno(0x5f)
1279      EOVERFLOW      = Errno(0x4b)
1280      EOWNERDEAD     = Errno(0x82)
1281      EPERM          = Errno(0x1)
1282      EPFNOSUPPORT   = Errno(0x60)
1283      EPIPE          = Errno(0x20)
1284      EPROTO         = Errno(0x47)
1285      EPROTONOSUPPORT = Errno(0x5d)
1286      EPROTOTYPE     = Errno(0x5b)
1287      ERANGE         = Errno(0x22)
1288      EREMCHG        = Errno(0x4e)
1289      EREMOTE        = Errno(0x42)
1290      EREMOTEIO      = Errno(0x79)
1291      ERESTART       = Errno(0x55)
1292      ERFKILL        = Errno(0x84)
1293      EROFS          = Errno(0x1e)
1294      ESHUTDOWN      = Errno(0x6c)
1295      ESOCKTNOSUPPORT = Errno(0x5e)
1296      ESPIPE         = Errno(0x1d)
1297      ESRCH          = Errno(0x3)
1298      ESRMNT         = Errno(0x45)
1299      ESTALE         = Errno(0x74)
1300      ESTRPIPE      = Errno(0x56)
1301      ETIME          = Errno(0x3e)
1302      ETIMEDOUT     = Errno(0x6e)
1303      ETOOMANYREFS  = Errno(0x6d)
1304      ETXTBSY       = Errno(0x1a)
1305      EUCLEAN        = Errno(0x75)
1306      EUNATCH        = Errno(0x31)
1307      EUSERS         = Errno(0x57)
1308      EWOULDBLOCK   = Errno(0xb)
1309      EXDEV          = Errno(0x12)
1310      EXFULL         = Errno(0x36)
1311 )
1312
1313 // Signals
1314 const (
1315     SIGABRT = Signal(0x6)
1316     SIGALRM = Signal(0xe)
1317     SIGBUS  = Signal(0x7)
1318     SIGCHLD = Signal(0x11)
1319     SIGCLD  = Signal(0x11)
1320     SIGCONT = Signal(0x12)
1321     SIGFPE  = Signal(0x8)
1322     SIGHUP  = Signal(0x1)
1323     SIGILL  = Signal(0x4)
1324     SIGINT  = Signal(0x2)

```

```
1325     SIGIO      = Signal(0x1d)
1326     SIGIOT     = Signal(0x6)
1327     SIGKILL    = Signal(0x9)
1328     SIGPIPE    = Signal(0xd)
1329     SIGPOLL    = Signal(0x1d)
1330     SIGPROF    = Signal(0x1b)
1331     SIGPWR     = Signal(0x1e)
1332     SIGQUIT    = Signal(0x3)
1333     SIGSEGV    = Signal(0xb)
1334     SIGSTKFLT  = Signal(0x10)
1335     SIGSTOP    = Signal(0x13)
1336     SIGSYS     = Signal(0x1f)
1337     SIGTERM    = Signal(0xf)
1338     SIGTRAP    = Signal(0x5)
1339     SIGTSTP    = Signal(0x14)
1340     SIGTTIN    = Signal(0x15)
1341     SIGTTOU    = Signal(0x16)
1342     SIGUNUSED  = Signal(0x1f)
1343     SIGURG     = Signal(0x17)
1344     SIGUSR1    = Signal(0xa)
1345     SIGUSR2    = Signal(0xc)
1346     SIGVTALRM  = Signal(0x1a)
1347     SIGWINCH   = Signal(0x1c)
1348     SIGXCPU    = Signal(0x18)
1349     SIGXFSZ    = Signal(0x19)
1350 )
1351
1352 // Error table
1353 var errors = [...]string{
1354     1: "operation not permitted",
1355     2: "no such file or directory",
1356     3: "no such process",
1357     4: "interrupted system call",
1358     5: "input/output error",
1359     6: "no such device or address",
1360     7: "argument list too long",
1361     8: "exec format error",
1362     9: "bad file descriptor",
1363     10: "no child processes",
1364     11: "resource temporarily unavailable",
1365     12: "cannot allocate memory",
1366     13: "permission denied",
1367     14: "bad address",
1368     15: "block device required",
1369     16: "device or resource busy",
1370     17: "file exists",
1371     18: "invalid cross-device link",
1372     19: "no such device",
1373     20: "not a directory",
```

1374	21:	"is a directory",
1375	22:	"invalid argument",
1376	23:	"too many open files in system",
1377	24:	"too many open files",
1378	25:	"inappropriate ioctl for device",
1379	26:	"text file busy",
1380	27:	"file too large",
1381	28:	"no space left on device",
1382	29:	"illegal seek",
1383	30:	"read-only file system",
1384	31:	"too many links",
1385	32:	"broken pipe",
1386	33:	"numerical argument out of domain",
1387	34:	"numerical result out of range",
1388	35:	"resource deadlock avoided",
1389	36:	"file name too long",
1390	37:	"no locks available",
1391	38:	"function not implemented",
1392	39:	"directory not empty",
1393	40:	"too many levels of symbolic links",
1394	42:	"no message of desired type",
1395	43:	"identifier removed",
1396	44:	"channel number out of range",
1397	45:	"level 2 not synchronized",
1398	46:	"level 3 halted",
1399	47:	"level 3 reset",
1400	48:	"link number out of range",
1401	49:	"protocol driver not attached",
1402	50:	"no CSI structure available",
1403	51:	"level 2 halted",
1404	52:	"invalid exchange",
1405	53:	"invalid request descriptor",
1406	54:	"exchange full",
1407	55:	"no anode",
1408	56:	"invalid request code",
1409	57:	"invalid slot",
1410	59:	"bad font file format",
1411	60:	"device not a stream",
1412	61:	"no data available",
1413	62:	"timer expired",
1414	63:	"out of streams resources",
1415	64:	"machine is not on the network",
1416	65:	"package not installed",
1417	66:	"object is remote",
1418	67:	"link has been severed",
1419	68:	"advertise error",
1420	69:	"srmount error",
1421	70:	"communication error on send",
1422	71:	"protocol error",
1423	72:	"multihop attempted",

1424 73: "RFS specific error",
1425 74: "bad message",
1426 75: "value too large for defined data type",
1427 76: "name not unique on network",
1428 77: "file descriptor in bad state",
1429 78: "remote address changed",
1430 79: "can not access a needed shared library",
1431 80: "accessing a corrupted shared library",
1432 81: ".lib section in a.out corrupted",
1433 82: "attempting to link in too many shared librarie",
1434 83: "cannot exec a shared library directly",
1435 84: "invalid or incomplete multibyte or wide charac",
1436 85: "interrupted system call should be restarted",
1437 86: "streams pipe error",
1438 87: "too many users",
1439 88: "socket operation on non-socket",
1440 89: "destination address required",
1441 90: "message too long",
1442 91: "protocol wrong type for socket",
1443 92: "protocol not available",
1444 93: "protocol not supported",
1445 94: "socket type not supported",
1446 95: "operation not supported",
1447 96: "protocol family not supported",
1448 97: "address family not supported by protocol",
1449 98: "address already in use",
1450 99: "cannot assign requested address",
1451 100: "network is down",
1452 101: "network is unreachable",
1453 102: "network dropped connection on reset",
1454 103: "software caused connection abort",
1455 104: "connection reset by peer",
1456 105: "no buffer space available",
1457 106: "transport endpoint is already connected",
1458 107: "transport endpoint is not connected",
1459 108: "cannot send after transport endpoint shutdown",
1460 109: "too many references: cannot splice",
1461 110: "connection timed out",
1462 111: "connection refused",
1463 112: "host is down",
1464 113: "no route to host",
1465 114: "operation already in progress",
1466 115: "operation now in progress",
1467 116: "stale NFS file handle",
1468 117: "structure needs cleaning",
1469 118: "not a XENIX named type file",
1470 119: "no XENIX semaphores available",
1471 120: "is a named type file",
1472 121: "remote I/O error",

```

1473         122: "disk quota exceeded",
1474         123: "no medium found",
1475         124: "wrong medium type",
1476         125: "operation canceled",
1477         126: "required key not available",
1478         127: "key has expired",
1479         128: "key has been revoked",
1480         129: "key was rejected by service",
1481         130: "owner died",
1482         131: "state not recoverable",
1483         132: "operation not possible due to RF-kill",
1484     }
1485
1486 // Signal table
1487 var signals = [...]string{
1488     1: "hangup",
1489     2: "interrupt",
1490     3: "quit",
1491     4: "illegal instruction",
1492     5: "trace/breakpoint trap",
1493     6: "aborted",
1494     7: "bus error",
1495     8: "floating point exception",
1496     9: "killed",
1497     10: "user defined signal 1",
1498     11: "segmentation fault",
1499     12: "user defined signal 2",
1500     13: "broken pipe",
1501     14: "alarm clock",
1502     15: "terminated",
1503     16: "stack fault",
1504     17: "child exited",
1505     18: "continued",
1506     19: "stopped (signal)",
1507     20: "stopped",
1508     21: "stopped (tty input)",
1509     22: "stopped (tty output)",
1510     23: "urgent I/O condition",
1511     24: "CPU time limit exceeded",
1512     25: "file size limit exceeded",
1513     26: "virtual timer expired",
1514     27: "profiling timer expired",
1515     28: "window changed",
1516     29: "I/O possible",
1517     30: "power failure",
1518     31: "bad system call",
1519 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/zsyscall_linux_amd64.g

```
1 // mksyscall.pl syscall_linux.go syscall_linux_amd64.go
2 // MACHINE GENERATED BY THE COMMAND ABOVE; DO NOT EDIT
3
4 package syscall
5
6 import "unsafe"
7
8 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
9
10 func open(path string, mode int, perm uint32) (fd int, err e
11     r0, _, e1 := Syscall(SYS_OPEN, uintptr(unsafe.Pointer
12     fd = int(r0)
13     if e1 != 0 {
14         err = e1
15     }
16     return
17 }
18
19 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
20
21 func openat(dirfd int, path string, flags int, mode uint32)
22     r0, _, e1 := Syscall6(SYS_OPENAT, uintptr(dirfd), ui
23     fd = int(r0)
24     if e1 != 0 {
25         err = e1
26     }
27     return
28 }
29
30 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
31
32 func pipe(p *[2]_C_int) (err error) {
33     _, _, e1 := RawSyscall(SYS_PIPE, uintptr(unsafe.Poin
34     if e1 != 0 {
35         err = e1
36     }
37     return
38 }
39
40 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
41
```

```

42 func utimes(path string, times *[2]Timeval) (err error) {
43     _, _, e1 := Syscall(SYS_UTIMES, uintptr(unsafe.Pointer(&path)),
44     if e1 != 0 {
45         err = e1
46     }
47     return
48 }
49
50 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
51
52 func futimesat(dirfd int, path *byte, times *[2]Timeval) (err error) {
53     _, _, e1 := Syscall(SYS_FUTIMESAT, uintptr(dirfd), uintptr(path),
54     if e1 != 0 {
55         err = e1
56     }
57     return
58 }
59
60 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
61
62 func Getcwd(buf []byte) (n int, err error) {
63     var _p0 unsafe.Pointer
64     if len(buf) > 0 {
65         _p0 = unsafe.Pointer(&buf[0])
66     } else {
67         _p0 = unsafe.Pointer(&_zero)
68     }
69     r0, _, e1 := Syscall(SYS_GETCWD, uintptr(_p0), uintptr(buf),
70     n = int(r0)
71     if e1 != 0 {
72         err = e1
73     }
74     return
75 }
76
77 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
78
79 func wait4(pid int, wstatus *_C_int, options int, rusage *Rusage) (wpid int, err error) {
80     r0, _, e1 := Syscall6(SYS_WAIT4, uintptr(pid), uintptr(wstatus),
81     wpid = int(r0)
82     if e1 != 0 {
83         err = e1
84     }
85     return
86 }
87
88 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
89
90 func ptrace(request int, pid int, addr uintptr, data uintptr) (err error) {
91     _, _, e1 := Syscall6(SYS_PTRACE, uintptr(request), uintptr(pid),

```

```

92         if e1 != 0 {
93             err = e1
94         }
95         return
96     }
97
98     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
99
100    func reboot(magic1 uint, magic2 uint, cmd int, arg string) (
101        _, _, e1 := Syscall6(SYS_REBOOT, uintptr(magic1), ui
102        if e1 != 0 {
103            err = e1
104        }
105        return
106    }
107
108    // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
109
110    func mount(source string, target string, fstype string, flag
111        _, _, e1 := Syscall6(SYS_MOUNT, uintptr(unsafe.Pointer
112        if e1 != 0 {
113            err = e1
114        }
115        return
116    }
117
118    // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
119
120    func Access(path string, mode uint32) (err error) {
121        _, _, e1 := Syscall(SYS_ACCESS, uintptr(unsafe.Pointer
122        if e1 != 0 {
123            err = e1
124        }
125        return
126    }
127
128    // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
129
130    func Acct(path string) (err error) {
131        _, _, e1 := Syscall(SYS_ACCT, uintptr(unsafe.Pointer
132        if e1 != 0 {
133            err = e1
134        }
135        return
136    }
137
138    // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
139
140    func Adjtimex(buf *Timex) (state int, err error) {

```

```

141         r0, _, e1 := Syscall(SYS_ADJTIMEX, uintptr(unsafe.Pointer(&state)),
142         state = int(r0)
143         if e1 != 0 {
144             err = e1
145         }
146         return
147     }
148
149 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
150
151 func Chdir(path string) (err error) {
152     _, _, e1 := Syscall(SYS_CHDIR, uintptr(unsafe.Pointer(&path)),
153     if e1 != 0 {
154         err = e1
155     }
156     return
157 }
158
159 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
160
161 func Chmod(path string, mode uint32) (err error) {
162     _, _, e1 := Syscall(SYS_CHMOD, uintptr(unsafe.Pointer(&path)),
163     if e1 != 0 {
164         err = e1
165     }
166     return
167 }
168
169 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
170
171 func Chroot(path string) (err error) {
172     _, _, e1 := Syscall(SYS_CHROOT, uintptr(unsafe.Pointer(&path)),
173     if e1 != 0 {
174         err = e1
175     }
176     return
177 }
178
179 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
180
181 func Close(fd int) (err error) {
182     _, _, e1 := Syscall(SYS_CLOSE, uintptr(fd), 0, 0)
183     if e1 != 0 {
184         err = e1
185     }
186     return
187 }
188
189 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT

```

```

190
191 func Creat(path string, mode uint32) (fd int, err error) {
192     r0, _, e1 := Syscall(SYS_CREAT, uintptr(unsafe.Pointer(
193     fd = int(r0)
194     if e1 != 0 {
195         err = e1
196     }
197     return
198 }
199
200 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
201
202 func Dup(oldfd int) (fd int, err error) {
203     r0, _, e1 := RawSyscall(SYS_DUP, uintptr(oldfd), 0,
204     fd = int(r0)
205     if e1 != 0 {
206         err = e1
207     }
208     return
209 }
210
211 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
212
213 func Dup2(oldfd int, newfd int) (err error) {
214     _, _, e1 := RawSyscall(SYS_DUP2, uintptr(oldfd), un
215     if e1 != 0 {
216         err = e1
217     }
218     return
219 }
220
221 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
222
223 func EpollCreate(size int) (fd int, err error) {
224     r0, _, e1 := RawSyscall(SYS_EPOLL_CREATE, uintptr(si
225     fd = int(r0)
226     if e1 != 0 {
227         err = e1
228     }
229     return
230 }
231
232 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
233
234 func EpollCreate1(flag int) (fd int, err error) {
235     r0, _, e1 := RawSyscall(SYS_EPOLL_CREATE1, uintptr(f
236     fd = int(r0)
237     if e1 != 0 {
238         err = e1
239     }

```

```

240         return
241     }
242
243 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
244
245 func EpollCtl(epfd int, op int, fd int, event *EpollEvent) (
246     _, _, e1 := RawSyscall6(SYS_EPOLL_CTL, uintptr(epfd)
247     if e1 != 0 {
248         err = e1
249     }
250     return
251 }
252
253 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
254
255 func EpollWait(epfd int, events []EpollEvent, msec int) (n i
256     var _p0 unsafe.Pointer
257     if len(events) > 0 {
258         _p0 = unsafe.Pointer(&events[0])
259     } else {
260         _p0 = unsafe.Pointer(&_zero)
261     }
262     r0, _, e1 := Syscall6(SYS_EPOLL_WAIT, uintptr(epfd),
263     n = int(r0)
264     if e1 != 0 {
265         err = e1
266     }
267     return
268 }
269
270 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
271
272 func Exit(code int) {
273     Syscall(SYS_EXIT_GROUP, uintptr(code), 0, 0)
274     return
275 }
276
277 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
278
279 func Faccessat(dirfd int, path string, mode uint32, flags in
280     _, _, e1 := Syscall6(SYS_FACCESSAT, uintptr(dirfd),
281     if e1 != 0 {
282         err = e1
283     }
284     return
285 }
286
287 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
288

```

```

289 func Fallocate(fd int, mode uint32, off int64, len int64) (e
290     _, _, e1 := Syscall6(SYS_FALLOCATE, uintptr(fd), uin
291     if e1 != 0 {
292         err = e1
293     }
294     return
295 }
296
297 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
298
299 func Fchdir(fd int) (err error) {
300     _, _, e1 := Syscall(SYS_FCHDIR, uintptr(fd), 0, 0)
301     if e1 != 0 {
302         err = e1
303     }
304     return
305 }
306
307 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
308
309 func Fchmod(fd int, mode uint32) (err error) {
310     _, _, e1 := Syscall(SYS_FCHMOD, uintptr(fd), uintptr
311     if e1 != 0 {
312         err = e1
313     }
314     return
315 }
316
317 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
318
319 func Fchmodat(dirfd int, path string, mode uint32, flags int
320     _, _, e1 := Syscall6(SYS_FCHMODAT, uintptr(dirfd), u
321     if e1 != 0 {
322         err = e1
323     }
324     return
325 }
326
327 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
328
329 func Fchownat(dirfd int, path string, uid int, gid int, flag
330     _, _, e1 := Syscall6(SYS_FCHOWNAT, uintptr(dirfd), u
331     if e1 != 0 {
332         err = e1
333     }
334     return
335 }
336
337 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT

```

```

338
339 func fcntl(fd int, cmd int, arg int) (val int, err error) {
340     r0, _, e1 := Syscall(SYS_FCNTL, uintptr(fd), uintptr
341     val = int(r0)
342     if e1 != 0 {
343         err = e1
344     }
345     return
346 }
347
348 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
349
350 func Fdatasync(fd int) (err error) {
351     _, _, e1 := Syscall(SYS_FDATASYNC, uintptr(fd), 0, 0
352     if e1 != 0 {
353         err = e1
354     }
355     return
356 }
357
358 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
359
360 func Flock(fd int, how int) (err error) {
361     _, _, e1 := Syscall(SYS_FLOCK, uintptr(fd), uintptr(
362     if e1 != 0 {
363         err = e1
364     }
365     return
366 }
367
368 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
369
370 func Fsync(fd int) (err error) {
371     _, _, e1 := Syscall(SYS_FSYNC, uintptr(fd), 0, 0)
372     if e1 != 0 {
373         err = e1
374     }
375     return
376 }
377
378 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
379
380 func Getdents(fd int, buf []byte) (n int, err error) {
381     var _p0 unsafe.Pointer
382     if len(buf) > 0 {
383         _p0 = unsafe.Pointer(&buf[0])
384     } else {
385         _p0 = unsafe.Pointer(&_zero)
386     }
387     r0, _, e1 := Syscall(SYS_GETDENTS64, uintptr(fd), ui

```

```

388         n = int(r0)
389         if e1 != 0 {
390             err = e1
391         }
392         return
393     }
394
395 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
396
397 func Getpgid(pid int) (pgid int, err error) {
398     r0, _, e1 := RawSyscall(SYS_GETPGID, uintptr(pid), 0
399     pgid = int(r0)
400     if e1 != 0 {
401         err = e1
402     }
403     return
404 }
405
406 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
407
408 func Getpgrp() (pid int) {
409     r0, _, _ := RawSyscall(SYS_GETPGRP, 0, 0, 0)
410     pid = int(r0)
411     return
412 }
413
414 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
415
416 func Getpid() (pid int) {
417     r0, _, _ := RawSyscall(SYS_GETPID, 0, 0, 0)
418     pid = int(r0)
419     return
420 }
421
422 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
423
424 func Getppid() (ppid int) {
425     r0, _, _ := RawSyscall(SYS_GETPPID, 0, 0, 0)
426     ppid = int(r0)
427     return
428 }
429
430 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
431
432 func Getrlimit(resource int, rlim *Rlimit) (err error) {
433     _, _, e1 := RawSyscall(SYS_GETRLIMIT, uintptr(resour
434     if e1 != 0 {
435         err = e1
436     }

```

```

437         return
438     }
439
440 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
441
442 func Getrusage(who int, rusage *Rusage) (err error) {
443     _, _, e1 := RawSyscall(SYS_GETRUSAGE, uintptr(who),
444     if e1 != 0 {
445         err = e1
446     }
447     return
448 }
449
450 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
451
452 func Gettid() (tid int) {
453     r0, _, _ := RawSyscall(SYS_GETTID, 0, 0, 0)
454     tid = int(r0)
455     return
456 }
457
458 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
459
460 func InotifyAddWatch(fd int, pathname string, mask uint32) (
461     r0, _, e1 := Syscall(SYS_INOTIFY_ADD_WATCH, uintptr(
462     watchdesc = int(r0)
463     if e1 != 0 {
464         err = e1
465     }
466     return
467 }
468
469 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
470
471 func InotifyInit() (fd int, err error) {
472     r0, _, e1 := RawSyscall(SYS_INOTIFY_INIT, 0, 0, 0)
473     fd = int(r0)
474     if e1 != 0 {
475         err = e1
476     }
477     return
478 }
479
480 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
481
482 func InotifyInit1(flags int) (fd int, err error) {
483     r0, _, e1 := RawSyscall(SYS_INOTIFY_INIT1, uintptr(f
484     fd = int(r0)
485     if e1 != 0 {

```

```

486             err = e1
487         }
488         return
489     }
490
491     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
492
493     func InotifyRmWatch(fd int, watchdesc uint32) (success int,
494         r0, _, e1 := RawSyscall(SYS_INOTIFY_RM_WATCH, uintpt
495         success = int(r0)
496         if e1 != 0 {
497             err = e1
498         }
499         return
500     }
501
502     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
503
504     func Kill(pid int, sig Signal) (err error) {
505         _, _, e1 := RawSyscall(SYS_KILL, uintptr(pid), uintp
506         if e1 != 0 {
507             err = e1
508         }
509         return
510     }
511
512     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
513
514     func Klogctl(typ int, buf []byte) (n int, err error) {
515         var _p0 unsafe.Pointer
516         if len(buf) > 0 {
517             _p0 = unsafe.Pointer(&buf[0])
518         } else {
519             _p0 = unsafe.Pointer(&_zero)
520         }
521         r0, _, e1 := Syscall(SYS_SYSLOG, uintptr(typ), uintp
522         n = int(r0)
523         if e1 != 0 {
524             err = e1
525         }
526         return
527     }
528
529     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
530
531     func Link(oldpath string, newpath string) (err error) {
532         _, _, e1 := Syscall(SYS_LINK, uintptr(unsafe.Pointer
533         if e1 != 0 {
534             err = e1
535         }

```

```

536         return
537     }
538
539 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
540
541 func Mkdir(path string, mode uint32) (err error) {
542     _, _, e1 := Syscall(SYS_MKDIR, uintptr(unsafe.Pointer(&path)),
543         uintptr(unsafe.Pointer(&mode)), 0)
544     if e1 != 0 {
545         err = e1
546     }
547     return
548 }
549 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
550
551 func Mkdirat(dirfd int, path string, mode uint32) (err error) {
552     _, _, e1 := Syscall(SYS_MKDIRAT, uintptr(dirfd), uintptr(unsafe.Pointer(&path)),
553         uintptr(unsafe.Pointer(&mode)), 0)
554     if e1 != 0 {
555         err = e1
556     }
557     return
558 }
559 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
560
561 func Mknod(path string, mode uint32, dev int) (err error) {
562     _, _, e1 := Syscall(SYS_MKNOD, uintptr(unsafe.Pointer(&path)),
563         uintptr(unsafe.Pointer(&mode)), uintptr(unsafe.Pointer(&dev)))
564     if e1 != 0 {
565         err = e1
566     }
567     return
568 }
569 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
570
571 func Mknodat(dirfd int, path string, mode uint32, dev int) (err error) {
572     _, _, e1 := Syscall6(SYS_MKNODAT, uintptr(dirfd), uintptr(unsafe.Pointer(&path)),
573         uintptr(unsafe.Pointer(&mode)), uintptr(unsafe.Pointer(&dev)), 0, 0)
574     if e1 != 0 {
575         err = e1
576     }
577     return
578 }
579 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
580
581 func Nanosleep(time *Timespec, leftover *Timespec) (err error) {
582     _, _, e1 := Syscall(SYS_NANOSLEEP, uintptr(unsafe.Pointer(&time)),
583         uintptr(unsafe.Pointer(&leftover)), 0)
584     if e1 != 0 {
585         err = e1
586     }
587 }

```

```

585         }
586         return
587     }
588
589     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
590
591     func Pause() (err error) {
592         _, _, e1 := Syscall(SYS_PAUSE, 0, 0, 0)
593         if e1 != 0 {
594             err = e1
595         }
596         return
597     }
598
599     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
600
601     func PivotRoot(newroot string, putold string) (err error) {
602         _, _, e1 := Syscall(SYS_PIVOT_ROOT, uintptr(unsafe.P
603         if e1 != 0 {
604             err = e1
605         }
606         return
607     }
608
609     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
610
611     func Read(fd int, p []byte) (n int, err error) {
612         var _p0 unsafe.Pointer
613         if len(p) > 0 {
614             _p0 = unsafe.Pointer(&p[0])
615         } else {
616             _p0 = unsafe.Pointer(&_zero)
617         }
618         r0, _, e1 := Syscall(SYS_READ, uintptr(fd), uintptr(
619         n = int(r0)
620         if e1 != 0 {
621             err = e1
622         }
623         return
624     }
625
626     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
627
628     func Readlink(path string, buf []byte) (n int, err error) {
629         var _p0 unsafe.Pointer
630         if len(buf) > 0 {
631             _p0 = unsafe.Pointer(&buf[0])
632         } else {
633             _p0 = unsafe.Pointer(&_zero)

```

```

634     }
635     r0, _, e1 := Syscall(SYS_READLINK, uintptr(unsafe.Pointer),
636     n = int(r0)
637     if e1 != 0 {
638         err = e1
639     }
640     return
641 }
642
643 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
644
645 func Rename(oldpath string, newpath string) (err error) {
646     _, _, e1 := Syscall(SYS_RENAME, uintptr(unsafe.Pointer),
647     if e1 != 0 {
648         err = e1
649     }
650     return
651 }
652
653 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
654
655 func Renameat.olddirfd int, oldpath string, newdirfd int, ne
656     _, _, e1 := Syscall6(SYS_RENAMEAT, uintptr(olddirfd),
657     if e1 != 0 {
658         err = e1
659     }
660     return
661 }
662
663 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
664
665 func Rmdir(path string) (err error) {
666     _, _, e1 := Syscall(SYS_RMDIR, uintptr(unsafe.Pointer),
667     if e1 != 0 {
668         err = e1
669     }
670     return
671 }
672
673 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
674
675 func Setdomainname(p []byte) (err error) {
676     var _p0 unsafe.Pointer
677     if len(p) > 0 {
678         _p0 = unsafe.Pointer(&p[0])
679     } else {
680         _p0 = unsafe.Pointer(&_zero)
681     }
682     _, _, e1 := Syscall(SYS_SETDOMAINNAME, uintptr(_p0),
683     if e1 != 0 {

```

```

684         err = e1
685     }
686     return
687 }
688
689 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
690
691 func Sethostname(p []byte) (err error) {
692     var _p0 unsafe.Pointer
693     if len(p) > 0 {
694         _p0 = unsafe.Pointer(&p[0])
695     } else {
696         _p0 = unsafe.Pointer(&_zero)
697     }
698     _, _, e1 := Syscall(SYS_SETHOSTNAME, uintptr(_p0), u
699     if e1 != 0 {
700         err = e1
701     }
702     return
703 }
704
705 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
706
707 func Setpgid(pid int, pgid int) (err error) {
708     _, _, e1 := RawSyscall(SYS_SETPGID, uintptr(pid), ui
709     if e1 != 0 {
710         err = e1
711     }
712     return
713 }
714
715 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
716
717 func Setrlimit(resource int, rlim *Rlimit) (err error) {
718     _, _, e1 := RawSyscall(SYS_SETRLIMIT, uintptr(resour
719     if e1 != 0 {
720         err = e1
721     }
722     return
723 }
724
725 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
726
727 func Setsid() (pid int, err error) {
728     r0, _, e1 := RawSyscall(SYS_SETSID, 0, 0, 0)
729     pid = int(r0)
730     if e1 != 0 {
731         err = e1
732     }

```

```

733         return
734     }
735
736 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
737
738 func Settimeofday(tv *Timeval) (err error) {
739     _, _, e1 := RawSyscall(SYS_SETTIMEOFDAY, uintptr(unsafe.Pointer(tv)), 0, 0)
740     if e1 != 0 {
741         err = e1
742     }
743     return
744 }
745
746 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
747
748 func Setuid(uid int) (err error) {
749     _, _, e1 := RawSyscall(SYS_SETUID, uintptr(uid), 0, 0)
750     if e1 != 0 {
751         err = e1
752     }
753     return
754 }
755
756 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
757
758 func Symlink(oldpath string, newpath string) (err error) {
759     _, _, e1 := Syscall(SYS_SYMLINK, uintptr(unsafe.Pointer(oldpath)), uintptr(unsafe.Pointer(newpath)), 0)
760     if e1 != 0 {
761         err = e1
762     }
763     return
764 }
765
766 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
767
768 func Sync() {
769     Syscall(SYS_SYNC, 0, 0, 0)
770     return
771 }
772
773 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
774
775 func Sysinfo(info *Sysinfo_t) (err error) {
776     _, _, e1 := RawSyscall(SYS_SYSINFO, uintptr(unsafe.Pointer(info)), 0, 0)
777     if e1 != 0 {
778         err = e1
779     }
780     return
781 }

```

```

782
783 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
784
785 func Tee(rfd int, wfd int, len int, flags int) (n int64, err
786     r0, _, e1 := Syscall6(SYS_TEE, uintptr(rfd), uintptr
787     n = int64(r0)
788     if e1 != 0 {
789         err = e1
790     }
791     return
792 }
793
794 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
795
796 func Tgkill(tgid int, tid int, sig Signal) (err error) {
797     _, _, e1 := RawSyscall(SYS_TGKILL, uintptr(tgid), ui
798     if e1 != 0 {
799         err = e1
800     }
801     return
802 }
803
804 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
805
806 func Times(tms *Tms) (ticks uintptr, err error) {
807     r0, _, e1 := RawSyscall(SYS_TIMES, uintptr(unsafe.Po
808     ticks = uintptr(r0)
809     if e1 != 0 {
810         err = e1
811     }
812     return
813 }
814
815 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
816
817 func Umask(mask int) (oldmask int) {
818     r0, _, _ := RawSyscall(SYS_UMASK, uintptr(mask), 0,
819     oldmask = int(r0)
820     return
821 }
822
823 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
824
825 func Uname(buf *Utsname) (err error) {
826     _, _, e1 := RawSyscall(SYS_UNAME, uintptr(unsafe.Poi
827     if e1 != 0 {
828         err = e1
829     }
830     return
831 }

```

```

832
833 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
834
835 func Unlink(path string) (err error) {
836     _, _, e1 := Syscall(SYS_UNLINK, uintptr(unsafe.Pointer(
837         path)), 0, 0)
838     if e1 != 0 {
839         err = e1
840     }
841     return
842 }
843 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
844
845 func Unlinkat(dirfd int, path string) (err error) {
846     _, _, e1 := Syscall(SYS_UNLINKAT, uintptr(dirfd), uintptr(
847         path), 0)
848     if e1 != 0 {
849         err = e1
850     }
851     return
852 }
853 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
854
855 func Unmount(target string, flags int) (err error) {
856     _, _, e1 := Syscall(SYS_UMOUNT2, uintptr(unsafe.Pointer(
857         target)), uintptr(flags), 0)
858     if e1 != 0 {
859         err = e1
860     }
861     return
862 }
863 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
864
865 func Unshare(flags int) (err error) {
866     _, _, e1 := Syscall(SYS_UNSHARE, uintptr(flags), 0,
867         0)
868     if e1 != 0 {
869         err = e1
870     }
871     return
872 }
873 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
874
875 func Ustat(dev int, ubuf *Ustat_t) (err error) {
876     _, _, e1 := Syscall(SYS_USTAT, uintptr(dev), uintptr(
877         ubuf), 0)
878     if e1 != 0 {
879         err = e1
880     }
881     return

```

```

881 }
882
883 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
884
885 func Utime(path string, buf *Utimbuf) (err error) {
886     _, _, e1 := Syscall(SYS_UTIME, uintptr(unsafe.Pointer(
887         buf)), 0, 0)
888     if e1 != 0 {
889         err = e1
890     }
891     return
892 }
893 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
894
895 func Write(fd int, p []byte) (n int, err error) {
896     var _p0 unsafe.Pointer
897     if len(p) > 0 {
898         _p0 = unsafe.Pointer(&p[0])
899     } else {
900         _p0 = unsafe.Pointer(&_zero)
901     }
902     r0, _, e1 := Syscall(SYS_WRITE, uintptr(fd), uintptr(
903         _p0), 0)
904     n = int(r0)
905     if e1 != 0 {
906         err = e1
907     }
908     return
909 }
910 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
911
912 func exitThread(code int) (err error) {
913     _, _, e1 := Syscall(SYS_EXIT, uintptr(code), 0, 0)
914     if e1 != 0 {
915         err = e1
916     }
917     return
918 }
919
920 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
921
922 func read(fd int, p *byte, np int) (n int, err error) {
923     r0, _, e1 := Syscall(SYS_READ, uintptr(fd), uintptr(
924         p), 0)
925     n = int(r0)
926     if e1 != 0 {
927         err = e1
928     }
929     return
930 }

```

```

930
931 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
932
933 func write(fd int, p *byte, np int) (n int, err error) {
934     r0, _, e1 := Syscall(SYS_WRITE, uintptr(fd), uintptr
935     n = int(r0)
936     if e1 != 0 {
937         err = e1
938     }
939     return
940 }
941
942 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
943
944 func munmap(addr uintptr, length uintptr) (err error) {
945     _, _, e1 := Syscall(SYS_MUNMAP, uintptr(addr), uintp
946     if e1 != 0 {
947         err = e1
948     }
949     return
950 }
951
952 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
953
954 func Madvise(b []byte, advice int) (err error) {
955     var _p0 unsafe.Pointer
956     if len(b) > 0 {
957         _p0 = unsafe.Pointer(&b[0])
958     } else {
959         _p0 = unsafe.Pointer(&_zero)
960     }
961     _, _, e1 := Syscall(SYS_MADVISE, uintptr(_p0), uintp
962     if e1 != 0 {
963         err = e1
964     }
965     return
966 }
967
968 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
969
970 func Mprotect(b []byte, prot int) (err error) {
971     var _p0 unsafe.Pointer
972     if len(b) > 0 {
973         _p0 = unsafe.Pointer(&b[0])
974     } else {
975         _p0 = unsafe.Pointer(&_zero)
976     }
977     _, _, e1 := Syscall(SYS_MPROTECT, uintptr(_p0), uint
978     if e1 != 0 {
979         err = e1

```

```

980         }
981     return
982 }
983
984 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
985
986 func Mlock(b []byte) (err error) {
987     var _p0 unsafe.Pointer
988     if len(b) > 0 {
989         _p0 = unsafe.Pointer(&b[0])
990     } else {
991         _p0 = unsafe.Pointer(&_zero)
992     }
993     _, _, e1 := Syscall(SYS_MLOCK, uintptr(_p0), uintptr(0), 0)
994     if e1 != 0 {
995         err = e1
996     }
997     return
998 }
999
1000 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1001
1002 func Munlock(b []byte) (err error) {
1003     var _p0 unsafe.Pointer
1004     if len(b) > 0 {
1005         _p0 = unsafe.Pointer(&b[0])
1006     } else {
1007         _p0 = unsafe.Pointer(&_zero)
1008     }
1009     _, _, e1 := Syscall(SYS_MUNLOCK, uintptr(_p0), uintptr(0), 0)
1010     if e1 != 0 {
1011         err = e1
1012     }
1013     return
1014 }
1015
1016 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1017
1018 func Mlockall(flags int) (err error) {
1019     _, _, e1 := Syscall(SYS_MLOCKALL, uintptr(flags), 0, 0)
1020     if e1 != 0 {
1021         err = e1
1022     }
1023     return
1024 }
1025
1026 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1027
1028 func Munlockall() (err error) {

```

```

1029         _, _, e1 := Syscall(SYS_MUNLOCKALL, 0, 0, 0)
1030     if e1 != 0 {
1031         err = e1
1032     }
1033     return
1034 }
1035
1036 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1037
1038 func Chown(path string, uid int, gid int) (err error) {
1039     _, _, e1 := Syscall(SYS_CHOWN, uintptr(unsafe.Pointer(
1040     if e1 != 0 {
1041         err = e1
1042     }
1043     return
1044 }
1045
1046 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1047
1048 func Fchown(fd int, uid int, gid int) (err error) {
1049     _, _, e1 := Syscall(SYS_FCHOWN, uintptr(fd), uintptr(
1050     if e1 != 0 {
1051         err = e1
1052     }
1053     return
1054 }
1055
1056 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1057
1058 func Fstat(fd int, stat *Stat_t) (err error) {
1059     _, _, e1 := Syscall(SYS_FSTAT, uintptr(fd), uintptr(
1060     if e1 != 0 {
1061         err = e1
1062     }
1063     return
1064 }
1065
1066 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1067
1068 func Fstatfs(fd int, buf *Statfs_t) (err error) {
1069     _, _, e1 := Syscall(SYS_FSTATFS, uintptr(fd), uintptr(
1070     if e1 != 0 {
1071         err = e1
1072     }
1073     return
1074 }
1075
1076 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1077

```

```

1078 func Ftruncate(fd int, length int64) (err error) {
1079     _, _, e1 := Syscall(SYS_FTRUNCATE, uintptr(fd), uint
1080     if e1 != 0 {
1081         err = e1
1082     }
1083     return
1084 }
1085
1086 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1087
1088 func Getegid() (egid int) {
1089     r0, _, _ := RawSyscall(SYS_GETEGID, 0, 0, 0)
1090     egid = int(r0)
1091     return
1092 }
1093
1094 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1095
1096 func Geteuid() (euid int) {
1097     r0, _, _ := RawSyscall(SYS_GETEUID, 0, 0, 0)
1098     euid = int(r0)
1099     return
1100 }
1101
1102 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1103
1104 func Getgid() (gid int) {
1105     r0, _, _ := RawSyscall(SYS_GETGID, 0, 0, 0)
1106     gid = int(r0)
1107     return
1108 }
1109
1110 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1111
1112 func Getuid() (uid int) {
1113     r0, _, _ := RawSyscall(SYS_GETUID, 0, 0, 0)
1114     uid = int(r0)
1115     return
1116 }
1117
1118 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1119
1120 func Ioperm(from int, num int, on int) (err error) {
1121     _, _, e1 := Syscall(SYS_IOPERM, uintptr(from), uintp
1122     if e1 != 0 {
1123         err = e1
1124     }
1125     return
1126 }
1127

```

```

1128 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1129
1130 func Iopl(level int) (err error) {
1131     _, _, e1 := Syscall(SYS_IOPL, uintptr(level), 0, 0)
1132     if e1 != 0 {
1133         err = e1
1134     }
1135     return
1136 }
1137
1138 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1139
1140 func Lchown(path string, uid int, gid int) (err error) {
1141     _, _, e1 := Syscall(SYS_LCHOWN, uintptr(unsafe.Pointer(
1142     if e1 != 0 {
1143         err = e1
1144     }
1145     return
1146 }
1147
1148 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1149
1150 func Listen(s int, n int) (err error) {
1151     _, _, e1 := Syscall(SYS_LISTEN, uintptr(s), uintptr(
1152     if e1 != 0 {
1153         err = e1
1154     }
1155     return
1156 }
1157
1158 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1159
1160 func Lstat(path string, stat *Stat_t) (err error) {
1161     _, _, e1 := Syscall(SYS_LSTAT, uintptr(unsafe.Pointer(
1162     if e1 != 0 {
1163         err = e1
1164     }
1165     return
1166 }
1167
1168 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1169
1170 func Pread(fd int, p []byte, offset int64) (n int, err error
1171     var _p0 unsafe.Pointer
1172     if len(p) > 0 {
1173         _p0 = unsafe.Pointer(&p[0])
1174     } else {
1175         _p0 = unsafe.Pointer(&_zero)
1176     }

```

```

1177         r0, _, e1 := Syscall6(SYS_PREAD64, uintptr(fd), uint
1178         n = int(r0)
1179         if e1 != 0 {
1180             err = e1
1181         }
1182         return
1183     }
1184
1185 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1186
1187 func Pwrite(fd int, p []byte, offset int64) (n int, err error) {
1188     var _p0 unsafe.Pointer
1189     if len(p) > 0 {
1190         _p0 = unsafe.Pointer(&p[0])
1191     } else {
1192         _p0 = unsafe.Pointer(&_zero)
1193     }
1194     r0, _, e1 := Syscall6(SYS_PWRITE64, uintptr(fd), uint
1195     n = int(r0)
1196     if e1 != 0 {
1197         err = e1
1198     }
1199     return
1200 }
1201
1202 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1203
1204 func Seek(fd int, offset int64, whence int) (off int64, err
1205     r0, _, e1 := Syscall(SYS_LSEEK, uintptr(fd), uintptr
1206     off = int64(r0)
1207     if e1 != 0 {
1208         err = e1
1209     }
1210     return
1211 }
1212
1213 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1214
1215 func Select(nfd int, r *FdSet, w *FdSet, e *FdSet, timeout *
1216     r0, _, e1 := Syscall6(SYS_SELECT, uintptr(nfd), uint
1217     n = int(r0)
1218     if e1 != 0 {
1219         err = e1
1220     }
1221     return
1222 }
1223
1224 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1225

```

```

1226 func Sendfile(outfd int, infd int, offset *int64, count int)
1227     r0, _, e1 := Syscall6(SYS_SENDFILE, uintptr(outfd),
1228     written = int(r0)
1229     if e1 != 0 {
1230         err = e1
1231     }
1232     return
1233 }
1234
1235 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1236
1237 func Setfsgid(gid int) (err error) {
1238     _, _, e1 := Syscall(SYS_SETFSGID, uintptr(gid), 0, 0
1239     if e1 != 0 {
1240         err = e1
1241     }
1242     return
1243 }
1244
1245 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1246
1247 func Setfsuid(uid int) (err error) {
1248     _, _, e1 := Syscall(SYS_SETFSUID, uintptr(uid), 0, 0
1249     if e1 != 0 {
1250         err = e1
1251     }
1252     return
1253 }
1254
1255 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1256
1257 func Setgid(gid int) (err error) {
1258     _, _, e1 := RawSyscall(SYS_SETGID, uintptr(gid), 0,
1259     if e1 != 0 {
1260         err = e1
1261     }
1262     return
1263 }
1264
1265 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1266
1267 func Setregid(rgid int, egid int) (err error) {
1268     _, _, e1 := RawSyscall(SYS_SETREGID, uintptr(rgid),
1269     if e1 != 0 {
1270         err = e1
1271     }
1272     return
1273 }
1274
1275 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT

```

```

1276
1277 func Setresgid(rgid int, egid int, sgid int) (err error) {
1278     _, _, e1 := RawSyscall(SYS_SETRESGID, uintptr(rgid),
1279     if e1 != 0 {
1280         err = e1
1281     }
1282     return
1283 }
1284
1285 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1286
1287 func Setresuid(ruid int, euid int, suid int) (err error) {
1288     _, _, e1 := RawSyscall(SYS_SETRESUID, uintptr(ruid),
1289     if e1 != 0 {
1290         err = e1
1291     }
1292     return
1293 }
1294
1295 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1296
1297 func Setreuid(ruid int, euid int) (err error) {
1298     _, _, e1 := RawSyscall(SYS_SETREUID, uintptr(ruid),
1299     if e1 != 0 {
1300         err = e1
1301     }
1302     return
1303 }
1304
1305 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1306
1307 func Shutdown(fd int, how int) (err error) {
1308     _, _, e1 := Syscall(SYS_SHUTDOWN, uintptr(fd), uintp
1309     if e1 != 0 {
1310         err = e1
1311     }
1312     return
1313 }
1314
1315 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1316
1317 func Splice(rfd int, roff *int64, wfd int, woff *int64, len
1318     r0, _, e1 := Syscall6(SYS_SPLICE, uintptr(rfd), uint
1319     n = int64(r0)
1320     if e1 != 0 {
1321         err = e1
1322     }
1323     return
1324 }

```

```

1325
1326 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1327
1328 func Stat(path string, stat *Stat_t) (err error) {
1329     _, _, e1 := Syscall(SYS_STAT, uintptr(unsafe.Pointer
1330     if e1 != 0 {
1331         err = e1
1332     }
1333     return
1334 }
1335
1336 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1337
1338 func Statfs(path string, buf *Statfs_t) (err error) {
1339     _, _, e1 := Syscall(SYS_STATFS, uintptr(unsafe.Point
1340     if e1 != 0 {
1341         err = e1
1342     }
1343     return
1344 }
1345
1346 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1347
1348 func SyncFileRange(fd int, off int64, n int64, flags int) (e
1349     _, _, e1 := Syscall6(SYS_SYNC_FILE_RANGE, uintptr(fd
1350     if e1 != 0 {
1351         err = e1
1352     }
1353     return
1354 }
1355
1356 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1357
1358 func Truncate(path string, length int64) (err error) {
1359     _, _, e1 := Syscall(SYS_TRUNCATE, uintptr(unsafe.Poi
1360     if e1 != 0 {
1361         err = e1
1362     }
1363     return
1364 }
1365
1366 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1367
1368 func accept(s int, rsa *RawSockaddrAny, addrlen *_Socklen) (
1369     r0, _, e1 := Syscall(SYS_ACCEPT, uintptr(s), uintptr
1370     fd = int(r0)
1371     if e1 != 0 {
1372         err = e1
1373     }

```

```

1374         return
1375     }
1376
1377 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1378
1379 func bind(s int, addr uintptr, addrlen _Socklen) (err error)
1380     _, _, e1 := Syscall(SYS_BIND, uintptr(s), uintptr(ad
1381     if e1 != 0 {
1382         err = e1
1383     }
1384     return
1385 }
1386
1387 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1388
1389 func connect(s int, addr uintptr, addrlen _Socklen) (err err
1390     _, _, e1 := Syscall(SYS_CONNECT, uintptr(s), uintptr
1391     if e1 != 0 {
1392         err = e1
1393     }
1394     return
1395 }
1396
1397 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1398
1399 func getgroups(n int, list *_Gid_t) (nn int, err error) {
1400     r0, _, e1 := RawSyscall(SYS_GETGROUPS, uintptr(n), u
1401     nn = int(r0)
1402     if e1 != 0 {
1403         err = e1
1404     }
1405     return
1406 }
1407
1408 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1409
1410 func setgroups(n int, list *_Gid_t) (err error) {
1411     _, _, e1 := RawSyscall(SYS_SETGROUPS, uintptr(n), ui
1412     if e1 != 0 {
1413         err = e1
1414     }
1415     return
1416 }
1417
1418 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1419
1420 func getsockopt(s int, level int, name int, val uintptr, val
1421     _, _, e1 := Syscall6(SYS_GETSOCKOPT, uintptr(s), uin
1422     if e1 != 0 {
1423         err = e1

```

```

1424     }
1425     return
1426 }
1427
1428 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1429
1430 func setsockopt(s int, level int, name int, val uintptr, val
1431     _, _, e1 := Syscall6(SYS_SETSOCKOPT, uintptr(s), uin
1432     if e1 != 0 {
1433         err = e1
1434     }
1435     return
1436 }
1437
1438 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1439
1440 func socket(domain int, typ int, proto int) (fd int, err err
1441     r0, _, e1 := RawSyscall(SYS_SOCKET, uintptr(domain),
1442     fd = int(r0)
1443     if e1 != 0 {
1444         err = e1
1445     }
1446     return
1447 }
1448
1449 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1450
1451 func socketpair(domain int, typ int, proto int, fd *[2]int)
1452     _, _, e1 := RawSyscall6(SYS_SOCKETPAIR, uintptr(doma
1453     if e1 != 0 {
1454         err = e1
1455     }
1456     return
1457 }
1458
1459 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1460
1461 func getpeername(fd int, rsa *RawSockaddrAny, addrlen *_Sock
1462     _, _, e1 := RawSyscall(SYS_GETPEERNAME, uintptr(fd),
1463     if e1 != 0 {
1464         err = e1
1465     }
1466     return
1467 }
1468
1469 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1470
1471 func getsockname(fd int, rsa *RawSockaddrAny, addrlen *_Sock
1472     _, _, e1 := RawSyscall(SYS_GETSOCKNAME, uintptr(fd),

```

```

1473         if e1 != 0 {
1474             err = e1
1475         }
1476         return
1477     }
1478
1479     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1480
1481     func recvfrom(fd int, p []byte, flags int, from *RawSockaddr
1482         var _p0 unsafe.Pointer
1483         if len(p) > 0 {
1484             _p0 = unsafe.Pointer(&p[0])
1485         } else {
1486             _p0 = unsafe.Pointer(&_zero)
1487         }
1488         r0, _, e1 := Syscall6(SYS_RECVFROM, uintptr(fd), uintp
1489         n = int(r0)
1490         if e1 != 0 {
1491             err = e1
1492         }
1493         return
1494     }
1495
1496     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1497
1498     func sendto(s int, buf []byte, flags int, to uintptr, addrle
1499         var _p0 unsafe.Pointer
1500         if len(buf) > 0 {
1501             _p0 = unsafe.Pointer(&buf[0])
1502         } else {
1503             _p0 = unsafe.Pointer(&_zero)
1504         }
1505         _, _, e1 := Syscall6(SYS_SENDDTO, uintptr(s), uintptr
1506         if e1 != 0 {
1507             err = e1
1508         }
1509         return
1510     }
1511
1512     // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1513
1514     func recvmsg(s int, msg *MsgHdr, flags int) (n int, err erro
1515         r0, _, e1 := Syscall(SYS_RECVMSG, uintptr(s), uintpt
1516         n = int(r0)
1517         if e1 != 0 {
1518             err = e1
1519         }
1520         return
1521     }

```

```
1522
1523 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1524
1525 func sendmsg(s int, msg *MsgHdr, flags int) (err error) {
1526     _, _, e1 := Syscall(SYS_SENDMSG, uintptr(s), uintptr
1527     if e1 != 0 {
1528         err = e1
1529     }
1530     return
1531 }
1532
1533 // THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT
1534
1535 func mmap(addr uintptr, length uintptr, prot int, flags int,
1536     r0, _, e1 := Syscall6(SYS_MMAP, uintptr(addr), uintp
1537     xaddr = uintptr(r0)
1538     if e1 != 0 {
1539         err = e1
1540     }
1541     return
1542 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/zsysnum_linux_amd64

```
1 // mksysnum_linux.pl /usr/include/asm/unistd_64.h
2 // MACHINE GENERATED BY THE ABOVE COMMAND; DO NOT EDIT
3
4 package syscall
5
6 const (
7     SYS_READ          = 0
8     SYS_WRITE        = 1
9     SYS_OPEN          = 2
10    SYS_CLOSE         = 3
11    SYS_STAT          = 4
12    SYS_FSTAT         = 5
13    SYS_LSTAT         = 6
14    SYS_POLL          = 7
15    SYS_LSEEK         = 8
16    SYS_MMAP          = 9
17    SYS_MPROTECT      = 10
18    SYS_MUNMAP        = 11
19    SYS_BRK           = 12
20    SYS_RT_SIGACTION  = 13
21    SYS_RT_SIGPROCMASK = 14
22    SYS_RT_SIGRETURN  = 15
23    SYS_IOCTL         = 16
24    SYS_PREAD64       = 17
25    SYS_PWRITE64      = 18
26    SYS_READV         = 19
27    SYS_WRITEV        = 20
28    SYS_ACCESS        = 21
29    SYS_PIPE          = 22
30    SYS_SELECT        = 23
31    SYS_SCHED_YIELD   = 24
32    SYS_MREMAP        = 25
33    SYS_MSYNC         = 26
34    SYS_MINCORE       = 27
35    SYS_MADVISE       = 28
36    SYS_SHMGET        = 29
37    SYS_SHMAT         = 30
38    SYS_SHMCTL        = 31
39    SYS_DUP           = 32
40    SYS_DUP2          = 33
41    SYS_PAUSE         = 34
```

42	SYS_NANOSLEEP	= 35
43	SYS_GETITIMER	= 36
44	SYS_ALARM	= 37
45	SYS_SETITIMER	= 38
46	SYS_GETPID	= 39
47	SYS_SENDFILE	= 40
48	SYS_SOCKET	= 41
49	SYS_CONNECT	= 42
50	SYS_ACCEPT	= 43
51	SYS_SENDTO	= 44
52	SYS_RECVFROM	= 45
53	SYS_SENDMSG	= 46
54	SYS_RECVMSG	= 47
55	SYS_SHUTDOWN	= 48
56	SYS_BIND	= 49
57	SYS_LISTEN	= 50
58	SYS_GETSOCKNAME	= 51
59	SYS_GETPEERNAME	= 52
60	SYS_SOCKETPAIR	= 53
61	SYS_SETSOCKOPT	= 54
62	SYS_GETSOCKOPT	= 55
63	SYS_CLONE	= 56
64	SYS_FORK	= 57
65	SYS_VFORK	= 58
66	SYS_EXECVE	= 59
67	SYS_EXIT	= 60
68	SYS_WAIT4	= 61
69	SYS_KILL	= 62
70	SYS_UNAME	= 63
71	SYS_SEMGET	= 64
72	SYS_SEMOP	= 65
73	SYS_SEMCTL	= 66
74	SYS_SHMDT	= 67
75	SYS_MSGGET	= 68
76	SYS_MSGSND	= 69
77	SYS_MSGRCV	= 70
78	SYS_MSGCTL	= 71
79	SYS_FCNTL	= 72
80	SYS_FLOCK	= 73
81	SYS_FSYNC	= 74
82	SYS_FDATASYNC	= 75
83	SYS_TRUNCATE	= 76
84	SYS_FTRUNCATE	= 77
85	SYS_GETDENTS	= 78
86	SYS_GETCWD	= 79
87	SYS_CHDIR	= 80
88	SYS_FCHDIR	= 81
89	SYS_RENAME	= 82
90	SYS_MKDIR	= 83
91	SYS_RMDIR	= 84

92	SYS_CREAT	= 85
93	SYS_LINK	= 86
94	SYS_UNLINK	= 87
95	SYS_SYMLINK	= 88
96	SYS_READLINK	= 89
97	SYS_CHMOD	= 90
98	SYS_FCHMOD	= 91
99	SYS_CHOWN	= 92
100	SYS_FCHOWN	= 93
101	SYS_LCHOWN	= 94
102	SYS_UMASK	= 95
103	SYS_GETTIMEOFDAY	= 96
104	SYS_GETRLIMIT	= 97
105	SYS_GETRUSAGE	= 98
106	SYS_SYSINFO	= 99
107	SYS_TIMES	= 100
108	SYS_PTRACE	= 101
109	SYS_GETUID	= 102
110	SYS_SYSLOG	= 103
111	SYS_GETGID	= 104
112	SYS_SETUID	= 105
113	SYS_SETGID	= 106
114	SYS_GETEUID	= 107
115	SYS_GETEGID	= 108
116	SYS_SETPGID	= 109
117	SYS_GETPPID	= 110
118	SYS_GETPGRP	= 111
119	SYS_SETSID	= 112
120	SYS_SETREUID	= 113
121	SYS_SETREGID	= 114
122	SYS_GETGROUPS	= 115
123	SYS_SETGROUPS	= 116
124	SYS_SETRESUID	= 117
125	SYS_GETRESUID	= 118
126	SYS_SETRESGID	= 119
127	SYS_GETRESGID	= 120
128	SYS_GETPGID	= 121
129	SYS_SETFSUID	= 122
130	SYS_SETFSGID	= 123
131	SYS_GETSID	= 124
132	SYS_CAPGET	= 125
133	SYS_CAPSET	= 126
134	SYS_RT_SIGPENDING	= 127
135	SYS_RT_SIGTIMEDWAIT	= 128
136	SYS_RT_SIGQUEUEINFO	= 129
137	SYS_RT_SIGSUSPEND	= 130
138	SYS_SIGALTSTACK	= 131
139	SYS_UTIME	= 132
140	SYS_MKNOD	= 133

141	SYS_USELIB	= 134
142	SYS_PERSONALITY	= 135
143	SYS_USTAT	= 136
144	SYS_STATFS	= 137
145	SYS_FSTATFS	= 138
146	SYS_SYSFS	= 139
147	SYS_GETPRIORITY	= 140
148	SYS_SETPRIORITY	= 141
149	SYS_SCHED_SETPARAM	= 142
150	SYS_SCHED_GETPARAM	= 143
151	SYS_SCHED_SETSCHEDULER	= 144
152	SYS_SCHED_GETSCHEDULER	= 145
153	SYS_SCHED_GET_PRIORITY_MAX	= 146
154	SYS_SCHED_GET_PRIORITY_MIN	= 147
155	SYS_SCHED_RR_GET_INTERVAL	= 148
156	SYS_MLOCK	= 149
157	SYS_MUNLOCK	= 150
158	SYS_MLOCKALL	= 151
159	SYS_MUNLOCKALL	= 152
160	SYS_VHANGUP	= 153
161	SYS_MODIFY_LDT	= 154
162	SYS_PIVOT_ROOT	= 155
163	SYS__SYSCTL	= 156
164	SYS_PRCTL	= 157
165	SYS_ARCH_PRCTL	= 158
166	SYS_ADJTIMEX	= 159
167	SYS_SETRLIMIT	= 160
168	SYS_CHROOT	= 161
169	SYS_SYNC	= 162
170	SYS_ACCT	= 163
171	SYS_SETTIMEOFDAY	= 164
172	SYS_MOUNT	= 165
173	SYS_UMOUNT2	= 166
174	SYS_SWAPON	= 167
175	SYS_SWAPOFF	= 168
176	SYS_REBOOT	= 169
177	SYS_SETHOSTNAME	= 170
178	SYS_SETDOMAINNAME	= 171
179	SYS_IOPL	= 172
180	SYS_IOPERM	= 173
181	SYS_CREATE_MODULE	= 174
182	SYS_INIT_MODULE	= 175
183	SYS_DELETE_MODULE	= 176
184	SYS_GET_KERNEL_SYMS	= 177
185	SYS_QUERY_MODULE	= 178
186	SYS_QUOTACTL	= 179
187	SYS_NFSSERVCTL	= 180
188	SYS_GETPMSG	= 181
189	SYS_PUTPMSG	= 182

190	SYS_AFS_SYSCALL	= 183
191	SYS_TUXCALL	= 184
192	SYS_SECURITY	= 185
193	SYS_GETTID	= 186
194	SYS_READAHEAD	= 187
195	SYS_SETXATTR	= 188
196	SYS_LSETXATTR	= 189
197	SYS_FSETXATTR	= 190
198	SYS_GETXATTR	= 191
199	SYS_LGETXATTR	= 192
200	SYS_FGETXATTR	= 193
201	SYS_LISTXATTR	= 194
202	SYS_LLISTXATTR	= 195
203	SYS_FLISTXATTR	= 196
204	SYS_REMOVEXATTR	= 197
205	SYS_LREMOVEXATTR	= 198
206	SYS_FREMOVEXATTR	= 199
207	SYS_TKILL	= 200
208	SYS_TIME	= 201
209	SYS_FUTEX	= 202
210	SYS_SCHED_SETAFFINITY	= 203
211	SYS_SCHED_GETAFFINITY	= 204
212	SYS_SET_THREAD_AREA	= 205
213	SYS_IO_SETUP	= 206
214	SYS_IO_DESTROY	= 207
215	SYS_IO_GETEVENTS	= 208
216	SYS_IO_SUBMIT	= 209
217	SYS_IO_CANCEL	= 210
218	SYS_GET_THREAD_AREA	= 211
219	SYS_LOOKUP_DCOOKIE	= 212
220	SYS_EPOLL_CREATE	= 213
221	SYS_EPOLL_CTL_OLD	= 214
222	SYS_EPOLL_WAIT_OLD	= 215
223	SYS_REMAP_FILE_PAGES	= 216
224	SYS_GETDENTS64	= 217
225	SYS_SET_TID_ADDRESS	= 218
226	SYS_RESTART_SYSCALL	= 219
227	SYS_SEMTIMEDOP	= 220
228	SYS_FADVISE64	= 221
229	SYS_TIMER_CREATE	= 222
230	SYS_TIMER_SETTIME	= 223
231	SYS_TIMER_GETTIME	= 224
232	SYS_TIMER_GETOVERRUN	= 225
233	SYS_TIMER_DELETE	= 226
234	SYS_CLOCK_SETTIME	= 227
235	SYS_CLOCK_GETTIME	= 228
236	SYS_CLOCK_GETRES	= 229
237	SYS_CLOCK_NANOSLEEP	= 230
238	SYS_EXIT_GROUP	= 231
239	SYS_EPOLL_WAIT	= 232

240	SYS_EPOLL_CTL	= 233
241	SYS_TGKILL	= 234
242	SYS_UTIMES	= 235
243	SYS_VSERVER	= 236
244	SYS_MBIND	= 237
245	SYS_SET_MEMPOLICY	= 238
246	SYS_GET_MEMPOLICY	= 239
247	SYS_MQ_OPEN	= 240
248	SYS_MQ_UNLINK	= 241
249	SYS_MQ_TIMEDSEND	= 242
250	SYS_MQ_TIMEDRECEIVE	= 243
251	SYS_MQ_NOTIFY	= 244
252	SYS_MQ_GETSETATTR	= 245
253	SYS_KEXEC_LOAD	= 246
254	SYS_WAITID	= 247
255	SYS_ADD_KEY	= 248
256	SYS_REQUEST_KEY	= 249
257	SYS_KEYCTL	= 250
258	SYS_IOPRIO_SET	= 251
259	SYS_IOPRIO_GET	= 252
260	SYS_INOTIFY_INIT	= 253
261	SYS_INOTIFY_ADD_WATCH	= 254
262	SYS_INOTIFY_RM_WATCH	= 255
263	SYS_MIGRATE_PAGES	= 256
264	SYS_OPENAT	= 257
265	SYS_MKDIRAT	= 258
266	SYS_MKNODAT	= 259
267	SYS_FCHOWNAT	= 260
268	SYS_FUTIMESAT	= 261
269	SYS_NEWFSTATAT	= 262
270	SYS_UNLINKAT	= 263
271	SYS_RENAMEAT	= 264
272	SYS_LINKAT	= 265
273	SYS_SYMLINKAT	= 266
274	SYS_READLINKAT	= 267
275	SYS_FCHMODAT	= 268
276	SYS_FACCESSAT	= 269
277	SYS_PSELECT6	= 270
278	SYS_PPOLL	= 271
279	SYS_UNSHARE	= 272
280	SYS_SET_ROBUST_LIST	= 273
281	SYS_GET_ROBUST_LIST	= 274
282	SYS_SPLICE	= 275
283	SYS_TEE	= 276
284	SYS_SYNC_FILE_RANGE	= 277
285	SYS_VMSPLICE	= 278
286	SYS_MOVE_PAGES	= 279
287	SYS_UTIMENSAT	= 280
288	SYS_EPOLL_PWAIT	= 281

289	SYS_SIGNALFD	= 282
290	SYS_TIMERFD_CREATE	= 283
291	SYS_EVENTFD	= 284
292	SYS_FALLOCATE	= 285
293	SYS_TIMERFD_SETTIME	= 286
294	SYS_TIMERFD_GETTIME	= 287
295	SYS_ACCEPT4	= 288
296	SYS_SIGNALFD4	= 289
297	SYS_EVENTFD2	= 290
298	SYS_EPOLL_CREATE1	= 291
299	SYS_DUP3	= 292
300	SYS_PIPE2	= 293
301	SYS_INOTIFY_INIT1	= 294
302	SYS_PREADV	= 295
303	SYS_PWRITEV	= 296
304	SYS_RT_TGSIGQUEUEINFO	= 297
305	SYS_PERF_EVENT_OPEN	= 298
306	SYS_RECVMSG	= 299
307	SYS_FANOTIFY_INIT	= 300
308	SYS_FANOTIFY_MARK	= 301
309	SYS_PRLIMIT64	= 302
310)	

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/syscall/ztypes_linux_amd64.go

```
1 // Created by cgo -godefs - DO NOT EDIT
2 // cgo -godefs types_linux.go
3
4 package syscall
5
6 const (
7     sizeofPtr      = 0x8
8     sizeofShort    = 0x2
9     sizeofInt      = 0x4
10    sizeofLong     = 0x8
11    sizeofLongLong = 0x8
12    PathMax        = 0x1000
13 )
14
15 type (
16     _C_short      int16
17     _C_int        int32
18     _C_long       int64
19     _C_long_long  int64
20 )
21
22 type Timespec struct {
23     Sec  int64
24     Nsec int64
25 }
26
27 type Timeval struct {
28     Sec  int64
29     Usec int64
30 }
31
32 type Timex struct {
33     Modes      uint32
34     Pad_cgo_0  [4]byte
35     Offset     int64
36     Freq       int64
37     Maxerror   int64
38     Esterror   int64
39     Status     int32
40     Pad_cgo_1  [4]byte
41     Constant   int64
```

```

42         Precision int64
43         Tolerance int64
44         Time      Timeval
45         Tick      int64
46         Ppsfreq  int64
47         Jitter   int64
48         Shift    int32
49         Pad_cgo_2 [4]byte
50         Stabil   int64
51         Jitcnt   int64
52         Calcnt   int64
53         Errcnt   int64
54         Stbcnt   int64
55         Tai      int32
56         Pad_cgo_3 [44]byte
57     }
58
59     type Time_t int64
60
61     type Tms struct {
62         Utime int64
63         Stime int64
64         Cutime int64
65         Cstime int64
66     }
67
68     type Utimbuf struct {
69         Actime int64
70         Modtime int64
71     }
72
73     type Rusage struct {
74         Utime      Timeval
75         Stime      Timeval
76         Maxrss    int64
77         Ixrss     int64
78         Idrss     int64
79         Isrss     int64
80         Minflt    int64
81         Majflt    int64
82         Nswap     int64
83         Inblock   int64
84         Oublock   int64
85         Msgsnd    int64
86         Msgrcv    int64
87         Nsignals  int64
88         Nvcsw     int64
89         Nivcsw    int64
90     }
91

```

```

92 type Rlimit struct {
93     Cur uint64
94     Max uint64
95 }
96
97 type _Gid_t uint32
98
99 type Stat_t struct {
100     Dev      uint64
101     Ino      uint64
102     Nlink    uint64
103     Mode     uint32
104     Uid      uint32
105     Gid      uint32
106     X__pad0  int32
107     Rdev     uint64
108     Size     int64
109     Blksize  int64
110     Blocks   int64
111     Atim     Timespec
112     Mtim     Timespec
113     Ctim     Timespec
114     X__unused [3]int64
115 }
116
117 type Statfs_t struct {
118     Type     int64
119     Bsize    int64
120     Blocks   uint64
121     Bfree    uint64
122     Bavail   uint64
123     Files    uint64
124     Ffree    uint64
125     Fsid     Fsid
126     Namelen  int64
127     Frsize   int64
128     Flags    int64
129     Spare    [4]int64
130 }
131
132 type Dirent struct {
133     Ino      uint64
134     Off      int64
135     Reclen   uint16
136     Type     uint8
137     Name     [256]int8
138     Pad_cgo_0 [5]byte
139 }
140

```

```
141 type Fsid struct {
142     X__val [2]int32
143 }
144
145 type RawSockaddrInet4 struct {
146     Family uint16
147     Port    uint16
148     Addr    [4]byte /* in_addr */
149     Zero    [8]uint8
150 }
151
152 type RawSockaddrInet6 struct {
153     Family    uint16
154     Port      uint16
155     Flowinfo  uint32
156     Addr      [16]byte /* in6_addr */
157     Scope_id  uint32
158 }
159
160 type RawSockaddrUnix struct {
161     Family uint16
162     Path   [108]int8
163 }
164
165 type RawSockaddrLinklayer struct {
166     Family    uint16
167     Protocol  uint16
168     Ifindex   int32
169     Hatype    uint16
170     Pktttype  uint8
171     Halen     uint8
172     Addr      [8]uint8
173 }
174
175 type RawSockaddrNetlink struct {
176     Family uint16
177     Pad    uint16
178     Pid    uint32
179     Groups uint32
180 }
181
182 type RawSockaddr struct {
183     Family uint16
184     Data   [14]int8
185 }
186
187 type RawSockaddrAny struct {
188     Addr RawSockaddr
189     Pad  [96]int8
```

```

190 }
191
192 type _Socklen uint32
193
194 type Linger struct {
195     Onoff int32
196     Linger int32
197 }
198
199 type Iovec struct {
200     Base *byte
201     Len uint64
202 }
203
204 type IPMreq struct {
205     Multiaddr [4]byte /* in_addr */
206     Interface [4]byte /* in_addr */
207 }
208
209 type IPMreqn struct {
210     Multiaddr [4]byte /* in_addr */
211     Address [4]byte /* in_addr */
212     Ifindex int32
213 }
214
215 type IPv6Mreq struct {
216     Multiaddr [16]byte /* in6_addr */
217     Interface uint32
218 }
219
220 type MsgHdr struct {
221     Name *byte
222     Namelen uint32
223     Pad_cgo_0 [4]byte
224     Iov *Iovec
225     Iovlen uint64
226     Control *byte
227     Controllen uint64
228     Flags int32
229     Pad_cgo_1 [4]byte
230 }
231
232 type Cmsghdr struct {
233     Len uint64
234     Level int32
235     Type int32
236     X__cmsg_data [0]byte
237 }
238
239 type Inet4Pktinfo struct {

```

```

240         Ifindex  int32
241         Spec_dst [4]byte /* in_addr */
242         Addr     [4]byte /* in_addr */
243     }
244
245     type Inet6Pktinfo struct {
246         Addr     [16]byte /* in6_addr */
247         Ifindex  uint32
248     }
249
250     type Ucred struct {
251         Pid int32
252         Uid uint32
253         Gid uint32
254     }
255
256     const (
257         SizeofSockaddrInet4   = 0x10
258         SizeofSockaddrInet6   = 0x1c
259         SizeofSockaddrAny     = 0x70
260         SizeofSockaddrUnix    = 0x6e
261         SizeofSockaddrLinklayer = 0x14
262         SizeofSockaddrNetlink = 0xc
263         SizeofLinger          = 0x8
264         SizeofIPMreq           = 0x8
265         SizeofIPMreqn          = 0xc
266         SizeofIPv6Mreq         = 0x14
267         SizeofMsghdr           = 0x38
268         SizeofCmsghdr          = 0x10
269         SizeofInet4Pktinfo     = 0xc
270         SizeofInet6Pktinfo     = 0x14
271         SizeofUcred             = 0xc
272     )
273
274     const (
275         IFA_UNSPEC      = 0x0
276         IFA_ADDRESS     = 0x1
277         IFA_LOCAL       = 0x2
278         IFA_LABEL       = 0x3
279         IFA_BROADCAST   = 0x4
280         IFA_ANYCAST     = 0x5
281         IFA_CACHEINFO   = 0x6
282         IFA_MULTICAST   = 0x7
283         IFLA_UNSPEC     = 0x0
284         IFLA_ADDRESS    = 0x1
285         IFLA_BROADCAST  = 0x2
286         IFLA_IFNAME     = 0x3
287         IFLA_MTU        = 0x4
288         IFLA_LINK       = 0x5

```

289	IFLA_QDISC	= 0x6
290	IFLA_STATS	= 0x7
291	IFLA_COST	= 0x8
292	IFLA_PRIORITY	= 0x9
293	IFLA_MASTER	= 0xa
294	IFLA_WIRELESS	= 0xb
295	IFLA_PROTINFO	= 0xc
296	IFLA_TXQLEN	= 0xd
297	IFLA_MAP	= 0xe
298	IFLA_WEIGHT	= 0xf
299	IFLA_OPERSTATE	= 0x10
300	IFLA_LINKMODE	= 0x11
301	IFLA_LINKINFO	= 0x12
302	IFLA_NET_NS_PID	= 0x13
303	IFLA_IFALIAS	= 0x14
304	IFLA_MAX	= 0x1c
305	RT_SCOPE_UNIVERSE	= 0x0
306	RT_SCOPE_SITE	= 0xc8
307	RT_SCOPE_LINK	= 0xfd
308	RT_SCOPE_HOST	= 0xfe
309	RT_SCOPE_NOWHERE	= 0xff
310	RT_TABLE_UNSPEC	= 0x0
311	RT_TABLE_COMPAT	= 0xfc
312	RT_TABLE_DEFAULT	= 0xfd
313	RT_TABLE_MAIN	= 0xfe
314	RT_TABLE_LOCAL	= 0xff
315	RT_TABLE_MAX	= 0xffffffff
316	RTA_UNSPEC	= 0x0
317	RTA_DST	= 0x1
318	RTA_SRC	= 0x2
319	RTA_IIF	= 0x3
320	RTA_OIF	= 0x4
321	RTA_GATEWAY	= 0x5
322	RTA_PRIORITY	= 0x6
323	RTA_PREFSRC	= 0x7
324	RTA_METRICS	= 0x8
325	RTA_MULTIPATH	= 0x9
326	RTA_FLOW	= 0xb
327	RTA_CACHEINFO	= 0xc
328	RTA_TABLE	= 0xf
329	RTN_UNSPEC	= 0x0
330	RTN_UNICAST	= 0x1
331	RTN_LOCAL	= 0x2
332	RTN_BROADCAST	= 0x3
333	RTN_ANYCAST	= 0x4
334	RTN_MULTICAST	= 0x5
335	RTN_BLACKHOLE	= 0x6
336	RTN_UNREACHABLE	= 0x7
337	RTN_PROHIBIT	= 0x8

```

338         RTN_THROW           = 0x9
339         RTN_NAT             = 0xa
340         RTN_XRESOLVE       = 0xb
341         SizeofNlMsgHdr     = 0x10
342         SizeofNlMsgerr     = 0x14
343         SizeofRtGenmsg     = 0x1
344         SizeofNlAttr       = 0x4
345         SizeofRtAttr       = 0x4
346         SizeofIfInfomsg   = 0x10
347         SizeofIfAddrmsg   = 0x8
348         SizeofRtMsg        = 0xc
349         SizeofRtNextHop   = 0x8
350     )
351
352     type NlMsgHdr struct {
353         Len      uint32
354         Type     uint16
355         Flags    uint16
356         Seq      uint32
357         Pid      uint32
358     }
359
360     type NlMsgerr struct {
361         Error    int32
362         Msg      NlMsgHdr
363     }
364
365     type RtGenmsg struct {
366         Family   uint8
367     }
368
369     type NlAttr struct {
370         Len      uint16
371         Type     uint16
372     }
373
374     type RtAttr struct {
375         Len      uint16
376         Type     uint16
377     }
378
379     type IfInfomsg struct {
380         Family   uint8
381         X__ifi_pad uint8
382         Type     uint16
383         Index    int32
384         Flags    uint32
385         Change   uint32
386     }
387

```

```

388 type IfAddrmsg struct {
389     Family    uint8
390     Prefixlen uint8
391     Flags     uint8
392     Scope     uint8
393     Index     uint32
394 }
395
396 type RtMsg struct {
397     Family    uint8
398     Dst_len   uint8
399     Src_len   uint8
400     Tos       uint8
401     Table     uint8
402     Protocol  uint8
403     Scope     uint8
404     Type      uint8
405     Flags     uint32
406 }
407
408 type RtNextHop struct {
409     Len       uint16
410     Flags     uint8
411     Hops      uint8
412     Ifindex   int32
413 }
414
415 const (
416     SizeofSockFilter = 0x8
417     SizeofSockFprog  = 0x10
418 )
419
420 type SockFilter struct {
421     Code uint16
422     Jt    uint8
423     Jf    uint8
424     K     uint32
425 }
426
427 type SockFprog struct {
428     Len       uint16
429     Pad_cgo_0 [6]byte
430     Filter    *SockFilter
431 }
432
433 type InotifyEvent struct {
434     Wd      int32
435     Mask    uint32
436     Cookie  uint32

```

```

437         Len    uint32
438         Name   [0]byte
439     }
440
441     const SizeofInotifyEvent = 0x10
442
443     type PtraceRegs struct {
444         R15    uint64
445         R14    uint64
446         R13    uint64
447         R12    uint64
448         Rbp    uint64
449         Rbx    uint64
450         R11    uint64
451         R10    uint64
452         R9     uint64
453         R8     uint64
454         Rax    uint64
455         Rcx    uint64
456         Rdx    uint64
457         Rsi    uint64
458         Rdi    uint64
459         Orig_rax uint64
460         Rip    uint64
461         Cs     uint64
462         Eflags uint64
463         Rsp    uint64
464         Ss     uint64
465         Fs_base uint64
466         Gs_base uint64
467         Ds     uint64
468         Es     uint64
469         Fs     uint64
470         Gs     uint64
471     }
472
473     type FdSet struct {
474         Bits [16]uint64
475     }
476
477     type Sysinfo_t struct {
478         Uptime    int64
479         Loads     [3]uint64
480         Totalram  uint64
481         Freeram   uint64
482         Sharedram uint64
483         Bufferram uint64
484         Totalswap uint64
485         Freeswap  uint64

```

```

486         Procs      uint16
487         Pad        uint16
488         Pad_cgo_0  [4]byte
489         Totalhigh  uint64
490         Freehigh   uint64
491         Unit        uint32
492         X_f         [0]byte
493         Pad_cgo_1  [4]byte
494     }
495
496     type Utsname struct {
497         Sysname      [65]int8
498         Nodename     [65]int8
499         Release      [65]int8
500         Version      [65]int8
501         Machine      [65]int8
502         Domainname  [65]int8
503     }
504
505     type Ustat_t struct {
506         Tfree        int32
507         Pad_cgo_0    [4]byte
508         Tinode       uint64
509         Fname        [6]int8
510         Fpack        [6]int8
511         Pad_cgo_1    [4]byte
512     }
513
514     type EpollEvent struct {
515         Events       uint32
516         Fd           int32
517         Pad          int32
518     }
519
520     type Termios struct {
521         Iflag        uint32
522         Oflag        uint32
523         Cflag        uint32
524         Lflag        uint32
525         Line         uint8
526         Cc           [32]uint8
527         Pad_cgo_0    [3]byte
528         Ispeed       uint32
529         Ospeed       uint32
530     }
531
532     const (
533         VINTR        = 0x0
534         VQUIT        = 0x1
535         VERASE       = 0x2

```

536	VKILL	= 0x3
537	VEOF	= 0x4
538	VTIME	= 0x5
539	VMIN	= 0x6
540	VSWTC	= 0x7
541	VSTART	= 0x8
542	VSTOP	= 0x9
543	VSUSP	= 0xa
544	VEOL	= 0xb
545	VREPRINT	= 0xc
546	VDISCARD	= 0xd
547	VWERASE	= 0xe
548	VLNEXT	= 0xf
549	VEOL2	= 0x10
550	IGNBRK	= 0x1
551	BRKINT	= 0x2
552	IGNPAR	= 0x4
553	PARMRK	= 0x8
554	INPCK	= 0x10
555	ISTRIP	= 0x20
556	INLCR	= 0x40
557	IGNCR	= 0x80
558	ICRNL	= 0x100
559	IUCLC	= 0x200
560	IXON	= 0x400
561	IXANY	= 0x800
562	IXOFF	= 0x1000
563	IMAXBEL	= 0x2000
564	IUTF8	= 0x4000
565	OPOST	= 0x1
566	OLCUC	= 0x2
567	ONLCR	= 0x4
568	OCRNL	= 0x8
569	ONOCR	= 0x10
570	ONLRET	= 0x20
571	OFILL	= 0x40
572	OFDEL	= 0x80
573	B0	= 0x0
574	B50	= 0x1
575	B75	= 0x2
576	B110	= 0x3
577	B134	= 0x4
578	B150	= 0x5
579	B200	= 0x6
580	B300	= 0x7
581	B600	= 0x8
582	B1200	= 0x9
583	B1800	= 0xa
584	B2400	= 0xb

```
585      B4800      = 0xc
586      B9600      = 0xd
587      B19200     = 0xe
588      B38400     = 0xf
589      CSIZE      = 0x30
590      CS5        = 0x0
591      CS6        = 0x10
592      CS7        = 0x20
593      CS8        = 0x30
594      CSTOPB     = 0x40
595      CREAD      = 0x80
596      PARENB     = 0x100
597      PARODD     = 0x200
598      HUPCL      = 0x400
599      CLOCAL     = 0x800
600      B57600    = 0x1001
601      B115200   = 0x1002
602      B230400   = 0x1003
603      B460800   = 0x1004
604      B500000   = 0x1005
605      B576000   = 0x1006
606      B921600   = 0x1007
607      B1000000  = 0x1008
608      B1152000  = 0x1009
609      B1500000  = 0x100a
610      B2000000  = 0x100b
611      B2500000  = 0x100c
612      B3000000  = 0x100d
613      B3500000  = 0x100e
614      B4000000  = 0x100f
615      ISIG       = 0x1
616      ICANON     = 0x2
617      XCASE      = 0x4
618      ECHO       = 0x8
619      ECHOE      = 0x10
620      ECHOK      = 0x20
621      ECHONL     = 0x40
622      NOFLSH    = 0x80
623      TOSTOP     = 0x100
624      ECHOCTL    = 0x200
625      ECHOPRT   = 0x400
626      ECHOKE    = 0x800
627      FLUSHO    = 0x1000
628      PENDIN    = 0x4000
629      IEXTEN     = 0x8000
630      TCGETS     = 0x5401
631      TCSETS     = 0x5402
632 )
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/testing/benchmark.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package testing
6
7 import (
8     "flag"
9     "fmt"
10    "os"
11    "runtime"
12    "time"
13 )
14
15 var matchBenchmarks = flag.String("test.bench", "", "regular
16 var benchTime = flag.Float64("test.benchtime", 1, "approxima
17
18 // An internal type but exported because it is cross-package
19 // of the "go test" command.
20 type InternalBenchmark struct {
21     Name string
22     F     func(b *B)
23 }
24
25 // B is a type passed to Benchmark functions to manage bench
26 // timing and to specify the number of iterations to run.
27 type B struct {
28     common
29     N          int
30     benchmark  InternalBenchmark
31     bytes      int64
32     timerOn    bool
33     result     BenchmarkResult
34 }
35
36 // StartTimer starts timing a test. This function is called
37 // before a benchmark starts, but it can also used to resume
38 // a call to StopTimer.
39 func (b *B) StartTimer() {
40     if !b.timerOn {
41         b.start = time.Now()
```

```

42             b.timerOn = true
43         }
44     }
45
46     // StopTimer stops timing a test. This can be used to pause
47     // while performing complex initialization that you don't
48     // want to measure.
49     func (b *B) StopTimer() {
50         if b.timerOn {
51             b.duration += time.Now().Sub(b.start)
52             b.timerOn = false
53         }
54     }
55
56     // ResetTimer sets the elapsed benchmark time to zero.
57     // It does not affect whether the timer is running.
58     func (b *B) ResetTimer() {
59         if b.timerOn {
60             b.start = time.Now()
61         }
62         b.duration = 0
63     }
64
65     // SetBytes records the number of bytes processed in a single
66     // If this is called, the benchmark will report ns/op and MB
67     func (b *B) SetBytes(n int64) { b.bytes = n }
68
69     func (b *B) nsPerOp() int64 {
70         if b.N <= 0 {
71             return 0
72         }
73         return b.duration.Nanoseconds() / int64(b.N)
74     }
75
76     // runN runs a single benchmark for the specified number of
77     func (b *B) runN(n int) {
78         // Try to get a comparable environment for each run
79         // by clearing garbage from previous runs.
80         runtime.GC()
81         b.N = n
82         b.ResetTimer()
83         b.StartTimer()
84         b.benchmark.F(b)
85         b.StopTimer()
86     }
87
88     func min(x, y int) int {
89         if x > y {
90             return y
91         }

```

```

92         return x
93     }
94
95     func max(x, y int) int {
96         if x < y {
97             return y
98         }
99         return x
100    }
101
102    // roundDown10 rounds a number down to the nearest power of
103    func roundDown10(n int) int {
104        var tens = 0
105        // tens = floor(log_10(n))
106        for n > 10 {
107            n = n / 10
108            tens++
109        }
110        // result = 10^tens
111        result := 1
112        for i := 0; i < tens; i++ {
113            result *= 10
114        }
115        return result
116    }
117
118    // roundUp rounds x up to a number of the form [1eX, 2eX, 5e
119    func roundUp(n int) int {
120        base := roundDown10(n)
121        if n < (2 * base) {
122            return 2 * base
123        }
124        if n < (5 * base) {
125            return 5 * base
126        }
127        return 10 * base
128    }
129
130    // run times the benchmark function in a separate goroutine.
131    func (b *B) run() BenchmarkResult {
132        go b.launch()
133        <-b.signal
134        return b.result
135    }
136
137    // launch launches the benchmark function. It gradually inc
138    // of benchmark iterations until the benchmark runs for a se
139    // to get a reasonable measurement. It prints timing inform
140    //           testing.BenchmarkHello 100000           19 n

```

```

141 // launch is run by the fun function as a separate goroutine
142 func (b *B) launch() {
143     // Run the benchmark for a single iteration in case
144     n := 1
145
146     // Signal that we're done whether we return normally
147     // or by FailNow's runtime.Goexit.
148     defer func() {
149         b.signal <- b
150     }()
151
152     b.runN(n)
153     // Run the benchmark for at least the specified amount
154     d := time.Duration(*benchTime * float64(time.Second))
155     for !b.failed && b.duration < d && n < 1e9 {
156         last := n
157         // Predict iterations/sec.
158         if b.nsPerOp() == 0 {
159             n = 1e9
160         } else {
161             n = int(d.Nanoseconds() / b.nsPerOp(
162         ))
163         // Run more iterations than we think we'll need
164         // Don't grow too fast in case we had timing
165         // Be sure to run at least one more than last
166         n = max(min(n+n/2, 100*last), last+1)
167         // Round up to something easy to read.
168         n = roundUp(n)
169         b.runN(n)
170     }
171     b.result = BenchmarkResult{b.N, b.duration, b.bytes}
172 }
173
174 // The results of a benchmark run.
175 type BenchmarkResult struct {
176     N      int           // The number of iterations.
177     T      time.Duration // The total time taken.
178     Bytes  int64        // Bytes processed in one iteration
179 }
180
181 func (r BenchmarkResult) NsPerOp() int64 {
182     if r.N <= 0 {
183         return 0
184     }
185     return r.T.Nanoseconds() / int64(r.N)
186 }
187
188 func (r BenchmarkResult) MbPerSec() float64 {
189     if r.Bytes <= 0 || r.T <= 0 || r.N <= 0 {

```

```

190         return 0
191     }
192     return (float64(r.Bytes) * float64(r.N) / 1e6) / r.T
193 }
194
195 func (r BenchmarkResult) String() string {
196     mbs := r.mbPerSec()
197     mb := ""
198     if mbs != 0 {
199         mb = fmt.Sprintf("\t%7.2f MB/s", mbs)
200     }
201     nsop := r.NsPerOp()
202     ns := fmt.Sprintf("%10d ns/op", nsop)
203     if r.N > 0 && nsop < 100 {
204         // The format specifiers here make sure that
205         // the ones digits line up for all three pos
206         if nsop < 10 {
207             ns = fmt.Sprintf("%13.2f ns/op", flo
208         } else {
209             ns = fmt.Sprintf("%12.1f ns/op", flo
210         }
211     }
212     return fmt.Sprintf("%8d\t%s%s", r.N, ns, mb)
213 }
214
215 // An internal function but exported because it is cross-pac
216 // of the "go test" command.
217 func RunBenchmarks(matchString func(pat, str string) (bool,
218     // If no flag was specified, don't run benchmarks.
219     if len(*matchBenchmarks) == 0 {
220         return
221     }
222     for _, Benchmark := range benchmarks {
223         matched, err := matchString(*matchBenchmarks
224         if err != nil {
225             fmt.Fprintf(os.Stderr, "testing: inv
226             os.Exit(1)
227         }
228         if !matched {
229             continue
230         }
231         for _, procs := range cpuList {
232             runtime.GOMAXPROCS(procs)
233             b := &B{
234                 common: common{
235                     signal: make(chan in
236                 },
237                 benchmark: Benchmark,
238             }
239             benchName := Benchmark.Name

```

```

240         if procs != 1 {
241             benchName = fmt.Sprintf("%s-
242         }
243         fmt.Printf("%s\t", benchName)
244         r := b.run()
245         if b.failed {
246             // The output could be very
247             // We print it all, regardle
248             // the benchmark failed.
249             fmt.Printf("--- FAIL: %s\n%s
250             continue
251         }
252         fmt.Printf("%v\n", r)
253         // Unlike with tests, we ignore the
254         // benchmarks since the output gener
255         if len(b.output) > 0 {
256             b.trimOutput()
257             fmt.Printf("--- BENCH: %s\n%
258         }
259         if p := runtime.GOMAXPROCS(-1); p !=
260             fmt.Fprintf(os.Stderr, "test
261         }
262     }
263 }
264 }
265
266 // trimOutput shortens the output from a benchmark, which ca
267 func (b *B) trimOutput() {
268     // The output is likely to appear multiple times bec
269     // is run multiple times, but at least it will be se
270     // because benchmarks rarely print, but just in case
271     const maxNewlines = 10
272     for nlCount, j := 0, 0; j < len(b.output); j++ {
273         if b.output[j] == '\n' {
274             nlCount++
275             if nlCount >= maxNewlines {
276                 b.output = append(b.output[:
277                 break
278             }
279         }
280     }
281 }
282
283 // Benchmark benchmarks a single function. Useful for creati
284 // custom benchmarks that do not use the "go test" command.
285 func Benchmark(f func(b *B)) BenchmarkResult {
286     b := &B{
287         common: common{
288             signal: make(chan interface{}),

```

```
289         },
290         benchmark: InternalBenchmark{""}, f},
291     }
292     return b.run()
293 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/testing/example.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package testing
6
7 import (
8     "bytes"
9     "fmt"
10    "io"
11    "os"
12    "strings"
13    "time"
14 )
15
16 type InternalExample struct {
17     Name    string
18     F       func()
19     Output  string
20 }
21
22 func RunExamples(matchString func(pat, str string) (bool, error)) {
23     ok = true
24
25     var eg InternalExample
26
27     stdout, stderr := os.Stdout, os.Stderr
28
29     for _, eg = range examples {
30         matched, err := matchString(*match, eg.Name)
31         if err != nil {
32             fmt.Fprintf(os.Stderr, "testing: inv
33                 os.Exit(1)
34         }
35         if !matched {
36             continue
37         }
38         if *chatty {
39             fmt.Printf("=== RUN: %s\n", eg.Name)
40         }
41
42         // capture stdout and stderr
43         r, w, err := os.Pipe()
44         if err != nil {
```

```

45         fmt.Fprintln(os.Stderr, err)
46         os.Exit(1)
47     }
48     os.Stdout, os.Stderr = w, w
49     outC := make(chan string)
50     go func() {
51         buf := new(bytes.Buffer)
52         _, err := io.Copy(buf, r)
53         if err != nil {
54             fmt.Fprintf(stderr, "testing
55                 os.Exit(1)
56         }
57         outC <- buf.String()
58     }()
59
60     // run example
61     t0 := time.Now()
62     eg.F()
63     dt := time.Now().Sub(t0)
64
65     // close pipe, restore stdout/stderr, get ou
66     w.Close()
67     os.Stdout, os.Stderr = stdout, stderr
68     out := <-outC
69
70     // report any errors
71     tstr := fmt.Sprintf("%.2f seconds", dt.Se
72     if g, e := strings.TrimSpace(out), strings.T
73         fmt.Printf("--- FAIL: %s %s\n", eg.N
74             eg.Name, tstr, g, e)
75         ok = false
76     } else if *chatty {
77         fmt.Printf("--- PASS: %s %s\n", eg.N
78     }
79     }
80
81     return
82 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/testing/testing.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package testing provides support for automated testing of
6 // It is intended to be used in concert with the ``go test''
7 // execution of any function of the form
8 //     func TestXxx(*testing.T)
9 // where Xxx can be any alphanumeric string (but the first 1
10 // [a-z]) and serves to identify the test routine.
11 // These TestXxx routines should be declared within the pack
12 //
13 // Functions of the form
14 //     func BenchmarkXxx(*testing.B)
15 // are considered benchmarks, and are executed by the "go te
16 // the -test.bench flag is provided.
17 //
18 // A sample benchmark function looks like this:
19 //     func BenchmarkHello(b *testing.B) {
20 //         for i := 0; i < b.N; i++ {
21 //             fmt.Sprintf("hello")
22 //         }
23 //     }
24 //
25 // The benchmark package will vary b.N until the benchmark f
26 // long enough to be timed reliably. The output
27 //     testing.BenchmarkHello    10000000    282 ns/op
28 // means that the loop ran 10000000 times at a speed of 282
29 //
30 // If a benchmark needs some expensive setup before running,
31 // may be stopped:
32 //     func BenchmarkBigLen(b *testing.B) {
33 //         b.StopTimer()
34 //         big := NewBig()
35 //         b.StartTimer()
36 //         for i := 0; i < b.N; i++ {
37 //             big.Len()
38 //         }
39 //     }
40 //
41 // The package also runs and verifies example code. Example
42 // include a concluding comment that begins with "Output:" a
43 // the standard output of the function when the tests are ru
44 // examples of an example:
```

```

45 //
46 //     func ExampleHello() {
47 //         fmt.Println("hello")
48 //         // Output: hello
49 //     }
50 //
51 //     func ExampleSalutations() {
52 //         fmt.Println("hello, and")
53 //         fmt.Println("goodbye")
54 //         // Output:
55 //         // hello, and
56 //         // goodbye
57 //     }
58 //
59 // Example functions without output comments are compiled by
60 //
61 // The naming convention to declare examples for a function
62 // method M on type T are:
63 //
64 //     func ExampleF() { ... }
65 //     func ExampleT() { ... }
66 //     func ExampleT_M() { ... }
67 //
68 // Multiple example functions for a type/function/method may
69 // append a distinct suffix to the name. The suffix must
70 // lower-case letter.
71 //
72 //     func ExampleF_suffix() { ... }
73 //     func ExampleT_suffix() { ... }
74 //     func ExampleT_M_suffix() { ... }
75 //
76 // The entire test file is presented as the example when it
77 // example function, at least one other function, type, vari
78 // declaration, and no test or benchmark functions.
79 package testing
80
81 import (
82     "flag"
83     "fmt"
84     "os"
85     "runtime"
86     "runtime/pprof"
87     "strconv"
88     "strings"
89     "time"
90 )
91
92 var (
93     // The short flag requests that tests run more quick
94     // is provided by test writers themselves. The test

```

```

95         // home. The all.bash installation script sets it t
96         // efficient, but by default the flag is off so a pl
97         // full test of the package.
98         short = flag.Bool("test.short", false, "run smaller
99
100        // Report as tests are run; default is silent for su
101        chatty      = flag.Bool("test.v", false, "verbose
102        match       = flag.String("test.run", "", "regula
103        memProfile   = flag.String("test.memprofile", "",
104        memProfileRate = flag.Int("test.memprofilerate", 0,
105        cpuProfile   = flag.String("test.cpuprofile", "",
106        timeout      = flag.Duration("test.timeout", 0, "i
107        cpuListStr   = flag.String("test.cpu", "", "comma-
108        parallel     = flag.Int("test.parallel", runtime.C
109
110        haveExamples bool // are there examples?
111
112        cpuList []int
113    )
114
115    // common holds the elements common between T and B and
116    // captures common methods such as Errorf.
117    type common struct {
118        output []byte // Output generated by test or be
119        failed bool // Test or benchmark has failed.
120        start time.Time // Time test or benchmark started
121        duration time.Duration
122        self interface{} // To be sent on signal ch
123        signal chan interface{} // Output for serial tests
124    }
125
126    // Short reports whether the -test.short flag is set.
127    func Short() bool {
128        return *short
129    }
130
131    // decorate inserts the final newline if needed and indentat
132    // If addFileLine is true, it also prefixes the string with
133    func decorate(s string, addFileLine bool) string {
134        if addFileLine {
135            _, file, line, ok := runtime.Caller(3) // de
136            if ok {
137                // Truncate file name at last file n
138                if index := strings.LastIndex(file,
139                    file = file[index+1:]
140            } else if index = strings.LastIndex(
141                file = file[index+1:]
142            }
143        } else {

```

```

144             file = "???"
145             line = 1
146         }
147         s = fmt.Sprintf("%s:%d: %s", file, line, s)
148     }
149     s = "\t" + s // Every line is indented at least one
150     n := len(s)
151     if n > 0 && s[n-1] != '\n' {
152         s += "\n"
153         n++
154     }
155     for i := 0; i < n-1; i++ { // -1 to avoid final newl
156         if s[i] == '\n' {
157             // Second and subsequent lines are i
158             return s[0:i+1] + "\t" + decorate(s[
159         }
160     }
161     return s
162 }
163
164 // T is a type passed to Test functions to manage test state
165 // Logs are accumulated during execution and dumped to stand
166 type T struct {
167     common
168     name          string // Name of test.
169     startParallel chan bool // Parallel tests will wait
170 }
171
172 // Fail marks the function as having failed but continues ex
173 func (c *common) Fail() { c.failed = true }
174
175 // Failed returns whether the function has failed.
176 func (c *common) Failed() bool { return c.failed }
177
178 // FailNow marks the function as having failed and stops its
179 // Execution will continue at the next test or benchmark.
180 func (c *common) FailNow() {
181     c.Fail()
182
183     // Calling runtime.Goexit will exit the goroutine, w
184     // will run the deferred functions in this goroutine
185     // which will eventually run the deferred lines in t
186     // which will signal to the test loop that this test
187     //
188     // A previous version of this code said:
189     //
190     //     c.duration = ...
191     //     c.signal <- c.self
192     //     runtime.Goexit()

```

```

193         //
194         // This previous version duplicated code (those line
195         // tRunner no matter what), but worse the goroutine
196         // implicit in runtime.Goexit was not guaranteed to
197         // before the test exited. If a test deferred an im
198         // function (like removing temporary files), there w
199         // it would run on a test failure. Because we send
200         // a top-of-stack deferred function now, we know tha
201         // only happens after any other stacked defers have
202         runtime.Goexit()
203     }
204
205     // log generates the output. It's always at the same stack d
206     func (c *common) log(s string) {
207         c.output = append(c.output, decorate(s, true)...)
208     }
209
210     // Log formats its arguments using default formatting, analo
211     // and records the text in the error log.
212     func (c *common) Log(args ...interface{}) { c.log(fmt.Sprint
213
214     // Logf formats its arguments according to the format, analo
215     // and records the text in the error log.
216     func (c *common) Logf(format string, args ...interface{}) {
217
218     // Error is equivalent to Log() followed by Fail().
219     func (c *common) Error(args ...interface{}) {
220         c.log(fmt.Sprintln(args...))
221         c.Fail()
222     }
223
224     // Errorf is equivalent to Logf() followed by Fail().
225     func (c *common) Errorf(format string, args ...interface{})
226         c.log(fmt.Sprintf(format, args...))
227         c.Fail()
228     }
229
230     // Fatal is equivalent to Log() followed by FailNow().
231     func (c *common) Fatal(args ...interface{}) {
232         c.log(fmt.Sprintln(args...))
233         c.FailNow()
234     }
235
236     // Fatalf is equivalent to Logf() followed by FailNow().
237     func (c *common) Fatalf(format string, args ...interface{})
238         c.log(fmt.Sprintf(format, args...))
239         c.FailNow()
240     }
241
242     // Parallel signals that this test is to be run in parallel

```

```

243 // other parallel tests in this CPU group.
244 func (t *T) Parallel() {
245     t.signal <- (*T)(nil) // Release main testing loop
246     <-t.startParallel    // Wait for serial tests to fi
247 }
248
249 // An internal type but exported because it is cross-package
250 // of the "go test" command.
251 type InternalTest struct {
252     Name string
253     F     func(*T)
254 }
255
256 func tRunner(t *T, test *InternalTest) {
257     t.start = time.Now()
258
259     // When this goroutine is done, either because test.
260     // returned normally or because a test failure trigg
261     // a call to runtime.Goexit, record the duration and
262     // a signal saying that the test is done.
263     defer func() {
264         t.duration = time.Now().Sub(t.start)
265         // If the test panicked, print any test outp
266         if err := recover(); err != nil {
267             t.report()
268             panic(err)
269         }
270         t.signal <- t
271     }()
272
273     test.F(t)
274 }
275
276 // An internal function but exported because it is cross-pac
277 // of the "go test" command.
278 func Main(matchString func(pat, str string) (bool, error), t
279     flag.Parse()
280     parseCpuList()
281
282     before()
283     startAlarm()
284     haveExamples = len(examples) > 0
285     testOk := RunTests(matchString, tests)
286     exampleOk := RunExamples(matchString, examples)
287     if !testOk || !exampleOk {
288         fmt.Println("FAIL")
289         os.Exit(1)
290     }
291     fmt.Println("PASS")

```

```

292         stopAlarm()
293         RunBenchmarks(matchString, benchmarks)
294         after()
295     }
296
297     func (t *T) report() {
298         tstr := fmt.Sprintf("%.2f seconds)", t.duration.Sec
299         format := "--- %s: %s %s\n%s"
300         if t.failed {
301             fmt.Printf(format, "FAIL", t.name, tstr, t.o
302         } else if *chatty {
303             fmt.Printf(format, "PASS", t.name, tstr, t.o
304         }
305     }
306
307     func RunTests(matchString func(pat, str string) (bool, error
308         ok = true
309         if len(tests) == 0 && !haveExamples {
310             fmt.Fprintln(os.Stderr, "testing: warning: n
311             return
312         }
313         for _, procs := range cpuList {
314             runtime.GOMAXPROCS(procs)
315             // We build a new channel tree for each run
316             // collector merges in one channel all the u
317             // If all tests pump to the same channel, a
318             // kicks off a goroutine that Fails, yet the
319             // which skews the counting.
320             var collector = make(chan interface{})
321
322             numParallel := 0
323             startParallel := make(chan bool)
324
325             for i := 0; i < len(tests); i++ {
326                 matched, err := matchString(*match,
327                 if err != nil {
328                     fmt.Fprintf(os.Stderr, "test
329                     os.Exit(1)
330                 }
331                 if !matched {
332                     continue
333                 }
334                 testName := tests[i].Name
335                 if procs != 1 {
336                     testName = fmt.Sprintf("%s-%
337                 }
338                 t := &T{
339                     common: common{
340                         signal: make(chan in

```

```

341         },
342         name:         testName,
343         startParallel: startParallel
344     }
345     t.self = t
346     if *chatty {
347         fmt.Printf("=== RUN %s\n", t
348     }
349     go tRunner(t, &tests[i])
350     out := (<-t.signal).(*T)
351     if out == nil { // Parallel run.
352         go func() {
353             collector <- <-t.sig
354         }()
355         numParallel++
356         continue
357     }
358     t.report()
359     ok = ok && !out.failed
360 }
361
362     running := 0
363     for numParallel+running > 0 {
364         if running < *parallel && numParalle
365             startParallel <- true
366             running++
367             numParallel--
368             continue
369         }
370         t := (<-collector).(*T)
371         t.report()
372         ok = ok && !t.failed
373         running--
374     }
375 }
376     return
377 }
378
379 // before runs before all testing.
380 func before() {
381     if *memProfileRate > 0 {
382         runtime.MemProfileRate = *memProfileRate
383     }
384     if *cpuProfile != "" {
385         f, err := os.Create(*cpuProfile)
386         if err != nil {
387             fmt.Fprintf(os.Stderr, "testing: %s"
388                 return
389         }
390         if err := pprof.StartCPUProfile(f); err != n

```

```

391             fmt.Fprintf(os.Stderr, "testing: can
392             f.Close()
393             return
394         }
395         // Could save f so after can call f.Close; n
396     }
397 }
398 }
399
400 // after runs after all testing.
401 func after() {
402     if *cpuProfile != "" {
403         pprof.StopCPUProfile() // flushes profile to
404     }
405     if *memProfile != "" {
406         f, err := os.Create(*memProfile)
407         if err != nil {
408             fmt.Fprintf(os.Stderr, "testing: %s"
409             return
410         }
411         if err = pprof.WriteHeapProfile(f); err != n
412             fmt.Fprintf(os.Stderr, "testing: can
413         }
414         f.Close()
415     }
416 }
417
418 var timer *time.Timer
419
420 // startAlarm starts an alarm if requested.
421 func startAlarm() {
422     if *timeout > 0 {
423         timer = time.AfterFunc(*timeout, alarm)
424     }
425 }
426
427 // stopAlarm turns off the alarm.
428 func stopAlarm() {
429     if *timeout > 0 {
430         timer.Stop()
431     }
432 }
433
434 // alarm is called if the timeout expires.
435 func alarm() {
436     panic("test timed out")
437 }
438
439 func parseCpuList() {

```

```
440     if len(*cpuListStr) == 0 {
441         cpuList = append(cpuList, runtime.GOMAXPROCS
442     } else {
443         for _, val := range strings.Split(*cpuListStr, ",") {
444             cpu, err := strconv.Atoi(val)
445             if err != nil || cpu <= 0 {
446                 fmt.Fprintf(os.Stderr, "test: invalid cpu %s\n", val)
447                 os.Exit(1)
448             }
449             cpuList = append(cpuList, cpu)
450         }
451     }
452 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/testing/iotest/logger.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package iotest
6
7 import (
8     "io"
9     "log"
10 )
11
12 type writeLogger struct {
13     prefix string
14     w      io.Writer
15 }
16
17 func (l *writeLogger) Write(p []byte) (n int, err error) {
18     n, err = l.w.Write(p)
19     if err != nil {
20         log.Printf("%s %x: %v", l.prefix, p[0:n], err)
21     } else {
22         log.Printf("%s %x", l.prefix, p[0:n])
23     }
24     return
25 }
26
27 // NewWriteLogger returns a writer that behaves like w except
28 // that it logs (using log.Printf) each write to standard er
29 // printing the prefix and the hexadecimal data written.
30 func NewWriteLogger(prefix string, w io.Writer) io.Writer {
31     return &writeLogger{prefix, w}
32 }
33
34 type readLogger struct {
35     prefix string
36     r      io.Reader
37 }
38
39 func (l *readLogger) Read(p []byte) (n int, err error) {
40     n, err = l.r.Read(p)
41     if err != nil {
```

```
42         log.Printf("%s %x: %v", l.prefix, p[0:n], er
43     } else {
44         log.Printf("%s %x", l.prefix, p[0:n])
45     }
46     return
47 }
48
49 // NewReadLogger returns a reader that behaves like r except
50 // that it logs (using log.Print) each read to standard erro
51 // printing the prefix and the hexadecimal data written.
52 func NewReadLogger(prefix string, r io.Reader) io.Reader {
53     return &readLogger{prefix, r}
54 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/testing/iotest/reader.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package iotest implements Readers and Writers useful main
6 package iotest
7
8 import (
9     "errors"
10    "io"
11 )
12
13 // OneByteReader returns a Reader that implements
14 // each non-empty Read by reading one byte from r.
15 func OneByteReader(r io.Reader) io.Reader { return &oneByteR
16
17 type oneByteReader struct {
18     r io.Reader
19 }
20
21 func (r *oneByteReader) Read(p []byte) (int, error) {
22     if len(p) == 0 {
23         return 0, nil
24     }
25     return r.r.Read(p[0:1])
26 }
27
28 // HalfReader returns a Reader that implements Read
29 // by reading half as many requested bytes from r.
30 func HalfReader(r io.Reader) io.Reader { return &halfReader{
31
32 type halfReader struct {
33     r io.Reader
34 }
35
36 func (r *halfReader) Read(p []byte) (int, error) {
37     return r.r.Read(p[0 : (len(p)+1)/2])
38 }
39
40 // DataErrReader returns a Reader that returns the final
41 // error with the last data read, instead of by itself with
```

```

42 // zero bytes of data.
43 func DataErrReader(r io.Reader) io.Reader { return &dataErrR
44
45 type dataErrReader struct {
46     r      io.Reader
47     unread []byte
48     data   []byte
49 }
50
51 func (r *dataErrReader) Read(p []byte) (n int, err error) {
52     // loop because first call needs two reads:
53     // one to get data and a second to look for an error
54     for {
55         if len(r.unread) == 0 {
56             n1, err1 := r.r.Read(r.data)
57             r.unread = r.data[0:n1]
58             err = err1
59         }
60         if n > 0 || err != nil {
61             break
62         }
63         n = copy(p, r.unread)
64         r.unread = r.unread[n:]
65     }
66     return
67 }
68
69 var ErrTimeout = errors.New("timeout")
70
71 // TimeoutReader returns ErrTimeout on the second read
72 // with no data. Subsequent calls to read succeed.
73 func TimeoutReader(r io.Reader) io.Reader { return &timeoutR
74
75 type timeoutReader struct {
76     r      io.Reader
77     count int
78 }
79
80 func (r *timeoutReader) Read(p []byte) (int, error) {
81     r.count++
82     if r.count == 2 {
83         return 0, ErrTimeout
84     }
85     return r.r.Read(p)
86 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/testing/iotest/writer.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package iotest
6
7 import "io"
8
9 // TruncateWriter returns a Writer that writes to w
10 // but stops silently after n bytes.
11 func TruncateWriter(w io.Writer, n int64) io.Writer {
12     return &truncateWriter{w, n}
13 }
14
15 type truncateWriter struct {
16     w io.Writer
17     n int64
18 }
19
20 func (t *truncateWriter) Write(p []byte) (n int, err error) {
21     if t.n <= 0 {
22         return len(p), nil
23     }
24     // real write
25     n = len(p)
26     if int64(n) > t.n {
27         n = int(t.n)
28     }
29     n, err = t.w.Write(p[0:n])
30     t.n -= int64(n)
31     if err == nil {
32         n = len(p)
33     }
34     return
35 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/testing/quick/quick.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package quick implements utility functions to help with b
6 package quick
7
8 import (
9     "flag"
10    "fmt"
11    "math"
12    "math/rand"
13    "reflect"
14    "strings"
15 )
16
17 var defaultMaxCount *int = flag.Int("quickchecks", 100, "The
18
19 // A Generator can generate random values of its own type.
20 type Generator interface {
21     // Generate returns a random instance of the type on
22     // method using the size as a size hint.
23     Generate(rand *rand.Rand, size int) reflect.Value
24 }
25
26 // randFloat32 generates a random float taking the full rang
27 func randFloat32(rand *rand.Rand) float32 {
28     f := rand.Float64() * math.MaxFloat32
29     if rand.Int()%1 == 1 {
30         f = -f
31     }
32     return float32(f)
33 }
34
35 // randFloat64 generates a random float taking the full rang
36 func randFloat64(rand *rand.Rand) float64 {
37     f := rand.Float64()
38     if rand.Int()%1 == 1 {
39         f = -f
40     }
41     return f
```

```

42 }
43
44 // randInt64 returns a random integer taking half the range
45 func randInt64(rand *rand.Rand) int64 { return rand.Int63()
46
47 // complexSize is the maximum length of arbitrary values tha
48 // values.
49 const complexSize = 50
50
51 // Value returns an arbitrary value of the given type.
52 // If the type implements the Generator interface, that will
53 // Note: To create arbitrary values for structs, all the fie
54 func Value(t reflect.Type, rand *rand.Rand) (value reflect.V
55     if m, ok := reflect.Zero(t).Interface().(Generator);
56         return m.Generate(rand, complexSize), true
57     }
58
59     switch concrete := t; concrete.Kind() {
60     case reflect.Bool:
61         return reflect.ValueOf(rand.Int()&1 == 0), t
62     case reflect.Float32:
63         return reflect.ValueOf(randFloat32(rand)), t
64     case reflect.Float64:
65         return reflect.ValueOf(randFloat64(rand)), t
66     case reflect.Complex64:
67         return reflect.ValueOf(complex(randFloat32(r
68     case reflect.Complex128:
69         return reflect.ValueOf(complex(randFloat64(r
70     case reflect.Int16:
71         return reflect.ValueOf(int16(randInt64(rand)
72     case reflect.Int32:
73         return reflect.ValueOf(int32(randInt64(rand)
74     case reflect.Int64:
75         return reflect.ValueOf(randInt64(rand)), tru
76     case reflect.Int8:
77         return reflect.ValueOf(int8(randInt64(rand))
78     case reflect.Int:
79         return reflect.ValueOf(int(randInt64(rand)))
80     case reflect.Uint16:
81         return reflect.ValueOf(uint16(randInt64(rand)
82     case reflect.Uint32:
83         return reflect.ValueOf(uint32(randInt64(rand)
84     case reflect.Uint64:
85         return reflect.ValueOf(uint64(randInt64(rand)
86     case reflect.Uint8:
87         return reflect.ValueOf(uint8(randInt64(rand)
88     case reflect.Uint:
89         return reflect.ValueOf(uint(randInt64(rand))
90     case reflect.Uintptr:
91         return reflect.ValueOf(uintptr(randInt64(ran

```

```

92     case reflect.Map:
93         numElems := rand.Intn(complexSize)
94         m := reflect.MakeMap(concrete)
95         for i := 0; i < numElems; i++ {
96             key, ok1 := Value(concrete.Key(), ra
97             value, ok2 := Value(concrete.Elem(),
98             if !ok1 || !ok2 {
99                 return reflect.Value{}, fals
100            }
101            m.SetMapIndex(key, value)
102        }
103        return m, true
104     case reflect.Ptr:
105         v, ok := Value(concrete.Elem(), rand)
106         if !ok {
107             return reflect.Value{}, false
108         }
109         p := reflect.New(concrete.Elem())
110         p.Elem().Set(v)
111         return p, true
112     case reflect.Slice:
113         numElems := rand.Intn(complexSize)
114         s := reflect.MakeSlice(concrete, numElems, n
115         for i := 0; i < numElems; i++ {
116             v, ok := Value(concrete.Elem(), rand
117             if !ok {
118                 return reflect.Value{}, fals
119             }
120             s.Index(i).Set(v)
121         }
122         return s, true
123     case reflect.String:
124         numChars := rand.Intn(complexSize)
125         codePoints := make([]rune, numChars)
126         for i := 0; i < numChars; i++ {
127             codePoints[i] = rune(rand.Intn(0x10f
128         }
129         return reflect.ValueOf(string(codePoints)),
130     case reflect.Struct:
131         s := reflect.New(t).Elem()
132         for i := 0; i < s.NumField(); i++ {
133             v, ok := Value(concrete.Field(i).Typ
134             if !ok {
135                 return reflect.Value{}, fals
136             }
137             s.Field(i).Set(v)
138         }
139         return s, true
140     default:

```

```

141         return reflect.Value{}, false
142     }
143
144     return
145 }
146
147 // A Config structure contains options for running a test.
148 type Config struct {
149     // MaxCount sets the maximum number of iterations. I
150     // MaxCountScale is used.
151     MaxCount int
152     // MaxCountScale is a non-negative scale factor appl
153     // maximum. If zero, the default is unchanged.
154     MaxCountScale float64
155     // If non-nil, rand is a source of random numbers. 0
156     // pseudo-random source will be used.
157     Rand *rand.Rand
158     // If non-nil, the Values function generates a slice
159     // reflect.Values that are congruent with the argume
160     // being tested. Otherwise, the top-level Values fun
161     // to generate them.
162     Values func([]reflect.Value, *rand.Rand)
163 }
164
165 var defaultConfig Config
166
167 // getRand returns the *rand.Rand to use for a given Config.
168 func (c *Config) getRand() *rand.Rand {
169     if c.Rand == nil {
170         return rand.New(rand.NewSource(0))
171     }
172     return c.Rand
173 }
174
175 // getMaxCount returns the maximum number of iterations to r
176 // Config.
177 func (c *Config) getMaxCount() (maxCount int) {
178     maxCount = c.MaxCount
179     if maxCount == 0 {
180         if c.MaxCountScale != 0 {
181             maxCount = int(c.MaxCountScale * flo
182         } else {
183             maxCount = *defaultMaxCount
184         }
185     }
186
187     return
188 }
189

```

```

190 // A SetupError is the result of an error in the way that ch
191 // used, independent of the functions being tested.
192 type SetupError string
193
194 func (s SetupError) Error() string { return string(s) }
195
196 // A CheckError is the result of Check finding an error.
197 type CheckError struct {
198     Count int
199     In    []interface{}
200 }
201
202 func (s *CheckError) Error() string {
203     return fmt.Sprintf("#%d: failed on input %s", s.Count, s.In)
204 }
205
206 // A CheckEqualError is the result CheckEqual finding an error.
207 type CheckEqualError struct {
208     CheckError
209     Out1 []interface{}
210     Out2 []interface{}
211 }
212
213 func (s *CheckEqualError) Error() string {
214     return fmt.Sprintf("#%d: failed on input %s. Output %s", s.Count, s.In, s.Out1)
215 }
216
217 // Check looks for an input to f, any function that returns
218 // such that f returns false. It calls f repeatedly, with a
219 // values for each argument. If f returns false on a given
220 // Check returns that input as a *CheckError.
221 // For example:
222 //
223 //     func TestOddMultipleOfThree(t *testing.T) {
224 //         f := func(x int) bool {
225 //             y := OddMultipleOfThree(x)
226 //             return y%2 == 1 && y%3 == 0
227 //         }
228 //         if err := quick.Check(f, nil); err != nil {
229 //             t.Error(err)
230 //         }
231 //     }
232 func Check(function interface{}, config *Config) (err error) {
233     if config == nil {
234         config = &defaultConfig
235     }
236
237     f, fType, ok := functionAndType(function)
238     if !ok {
239         err = SetupError("argument is not a function")

```

```

240         return
241     }
242
243     if fType.NumOut() != 1 {
244         err = SetupError("function returns more than
245             return
246     }
247     if fType.Out(0).Kind() != reflect.Bool {
248         err = SetupError("function does not return a
249             return
250     }
251
252     arguments := make([]reflect.Value, fType.NumIn())
253     rand := config.getRand()
254     maxCount := config.getMaxCount()
255
256     for i := 0; i < maxCount; i++ {
257         err = arbitraryValues(arguments, fType, conf
258             if err != nil {
259                 return
260             }
261
262             if !f.Call(arguments)[0].Bool() {
263                 err = &CheckError{i + 1, toInterface
264                     return
265             }
266     }
267
268     return
269 }
270
271 // CheckEqual looks for an input on which f and g return dif
272 // It calls f and g repeatedly with arbitrary values for eac
273 // If f and g return different answers, CheckEqual returns a
274 // describing the input and the outputs.
275 func CheckEqual(f, g interface{}, config *Config) (err error
276     if config == nil {
277         config = &defaultConfig
278     }
279
280     x, xType, ok := functionAndType(f)
281     if !ok {
282         err = SetupError("f is not a function")
283         return
284     }
285     y, yType, ok := functionAndType(g)
286     if !ok {
287         err = SetupError("g is not a function")
288         return

```

```

289     }
290
291     if xType != yType {
292         err = SetupError("functions have different t
293         return
294     }
295
296     arguments := make([]reflect.Value, xType.NumIn())
297     rand := config.getRand()
298     maxCount := config.getMaxCount()
299
300     for i := 0; i < maxCount; i++ {
301         err = arbitraryValues(arguments, xType, conf
302         if err != nil {
303             return
304         }
305
306         xOut := toInterfaces(x.Call(arguments))
307         yOut := toInterfaces(y.Call(arguments))
308
309         if !reflect.DeepEqual(xOut, yOut) {
310             err = &CheckEqualError{CheckError{i
311             return
312         }
313     }
314
315     return
316 }
317
318 // arbitraryValues writes Values to args such that args cont
319 // suitable for calling f.
320 func arbitraryValues(args []reflect.Value, f reflect.Type, c
321     if config.Values != nil {
322         config.Values(args, rand)
323         return
324     }
325
326     for j := 0; j < len(args); j++ {
327         var ok bool
328         args[j], ok = Value(f.In(j), rand)
329         if !ok {
330             err = SetupError(fmt.Sprintf("cannot
331             return
332         }
333     }
334
335     return
336 }
337

```

```

338 func functionAndType(f interface{}) (v reflect.Value, t refl
339     v = reflect.ValueOf(f)
340     ok = v.Kind() == reflect.Func
341     if !ok {
342         return
343     }
344     t = v.Type()
345     return
346 }
347
348 func toInterfaces(values []reflect.Value) []interface{} {
349     ret := make([]interface{}, len(values))
350     for i, v := range values {
351         ret[i] = v.Interface()
352     }
353     return ret
354 }
355
356 func toString(interfaces []interface{}) string {
357     s := make([]string, len(interfaces))
358     for i, v := range interfaces {
359         s[i] = fmt.Sprintf("%#v", v)
360     }
361     return strings.Join(s, ", ")
362 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/scanner/scanner.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package scanner provides a scanner and tokenizer for UTF-
6 // It takes an io.Reader providing the source, which then ca
7 // through repeated calls to the Scan function. For compati
8 // existing tools, the NUL character is not allowed.
9 //
10 // By default, a Scanner skips white space and Go comments a
11 // literals as defined by the Go language specification. It
12 // customized to recognize only a subset of those literals a
13 // different white space characters.
14 //
15 // Basic usage pattern:
16 //
17 //     var s scanner.Scanner
18 //     s.Init(src)
19 //     tok := s.Scan()
20 //     for tok != scanner.EOF {
21 //         // do something with tok
22 //         tok = s.Scan()
23 //     }
24 //
25 package scanner
26
27 import (
28     "bytes"
29     "fmt"
30     "io"
31     "os"
32     "unicode"
33     "unicode/utf8"
34 )
35
36 // TODO(gri): Consider changing this to use the new (token)
37
38 // A source position is represented by a Position value.
39 // A position is valid if Line > 0.
40 type Position struct {
41     Filename string // filename, if any
```

```

42         Offset    int    // byte offset, starting at 0
43         Line      int    // line number, starting at 1
44         Column    int    // column number, starting at 1 (cha
45     }
46
47     // IsValid returns true if the position is valid.
48     func (pos *Position) IsValid() bool { return pos.Line > 0 }
49
50     func (pos Position) String() string {
51         s := pos.FileName
52         if pos.IsValid() {
53             if s != "" {
54                 s += ":"
55             }
56             s += fmt.Sprintf("%d:%d", pos.Line, pos.Colu
57         }
58         if s == "" {
59             s = "???"
60         }
61         return s
62     }
63
64     // Predefined mode bits to control recognition of tokens. Fo
65     // to configure a Scanner such that it only recognizes (Go)
66     // integers, and skips comments, set the Scanner's Mode fiel
67     //
68     //     ScanIdents | ScanInts | SkipComments
69     //
70     const (
71         ScanIdents    = 1 << -Ident
72         ScanInts      = 1 << -Int
73         ScanFloats    = 1 << -Float // includes Ints
74         ScanChars     = 1 << -Char
75         ScanStrings   = 1 << -String
76         ScanRawStrings = 1 << -RawString
77         ScanComments  = 1 << -Comment
78         SkipComments  = 1 << -skipComment // if set with Sc
79         GoTokens      = ScanIdents | ScanFloats | ScanChars
80     )
81
82     // The result of Scan is one of the following tokens or a Un
83     const (
84         EOF = -(iota + 1)
85         Ident
86         Int
87         Float
88         Char
89         String
90         RawString
91         Comment

```

```

92         skipComment
93     )
94
95     var tokenString = map[rune]string{
96         EOF:      "EOF",
97         Ident:    "Ident",
98         Int:      "Int",
99         Float:    "Float",
100        Char:     "Char",
101        String:   "String",
102        RawString: "RawString",
103        Comment:  "Comment",
104    }
105
106    // TokenString returns a printable string for a token or Uni
107    func TokenString(tok rune) string {
108        if s, found := tokenString[tok]; found {
109            return s
110        }
111        return fmt.Sprintf("%q", string(tok))
112    }
113
114    // GoWhitespace is the default value for the Scanner's White
115    // Its value selects Go's white space characters.
116    const GoWhitespace = 1<<' \t' | 1<<' \n' | 1<<' \r' | 1<<' '
117
118    const bufLen = 1024 // at least utf8.UTFMax
119
120    // A Scanner implements reading of Unicode characters and to
121    type Scanner struct {
122        // Input
123        src io.Reader
124
125        // Source buffer
126        srcBuf [bufLen + 1]byte // +1 for sentinel for commo
127        srcPos int             // reading position (srcBuf
128        srcEnd int             // source end (srcBuf index)
129
130        // Source position
131        srcBufOffset int // byte offset of srcBuf[0] in sour
132        line          int // line count
133        column        int // character count
134        lastLineLen  int // length of last line in character
135        lastCharLen  int // length of last character in byte
136
137        // Token text buffer
138        // Typically, token text is stored completely in src
139        // the token text's head may be buffered in tokBuf w
140        // tail is stored in srcBuf.

```

```

141     tokBuf bytes.Buffer // token text head that is not i
142     tokPos int          // token text tail position (src
143     tokEnd int         // token text tail end (srcBuf i
144
145     // One character look-ahead
146     ch rune // character before current srcPos
147
148     // Error is called for each error encountered. If no
149     // function is set, the error is reported to os.Stde
150     Error func(s *Scanner, msg string)
151
152     // ErrorCount is incremented by one for each error e
153     ErrorCount int
154
155     // The Mode field controls which tokens are recogniz
156     // to recognize Ints, set the ScanInts bit in Mode.
157     // changed at any time.
158     Mode uint
159
160     // The Whitespace field controls which characters ar
161     // as white space. To recognize a character ch <= '
162     // set the ch'th bit in Whitespace (the Scanner's be
163     // for values ch > ' '). The field may be changed at
164     Whitespace uint64
165
166     // Start position of most recently scanned token; se
167     // Calling Init or Next invalidates the position (Li
168     // The Filename field is always left untouched by th
169     // If an error is reported (via Error) and Position
170     // the scanner is not inside a token. Call Pos to ob
171     // position in that case.
172     Position
173 }
174
175 // Init initializes a Scanner with a new source and returns
176 // Error is set to nil, ErrorCount is set to 0, Mode is set
177 // and Whitespace is set to GoWhitespace.
178 func (s *Scanner) Init(src io.Reader) *Scanner {
179     s.src = src
180
181     // initialize source buffer
182     // (the first call to next() will fill it by calling
183     s.srcBuf[0] = utf8.RuneSelf // sentinel
184     s.srcPos = 0
185     s.srcEnd = 0
186
187     // initialize source position
188     s.srcBufOffset = 0
189     s.line = 1

```

```

190         s.column = 0
191         s.lastLineLen = 0
192         s.lastCharLen = 0
193
194         // initialize token text buffer
195         // (required for first call to next()).
196         s.tokPos = -1
197
198         // initialize one character look-ahead
199         s.ch = -1 // no char read yet
200
201         // initialize public fields
202         s.Error = nil
203         s.ErrorCount = 0
204         s.Mode = GoTokens
205         s.Whitespace = GoWhitespace
206         s.Line = 0 // invalidate token position
207
208         return s
209     }
210
211     // TODO(gri): The code for next() and the internal scanner s
212     //             from a rethink. While next() is optimized for
213     //             case, the "corrections" needed for proper posi
214     //             some of the attempts for fast-path optimizatio
215
216     // next reads and returns the next Unicode character. It is
217     // that only a minimal amount of work needs to be done in th
218     // case (one test to check for both ASCII and end-of-buffer,
219     // to check for newlines).
220     func (s *Scanner) next() rune {
221         ch, width := rune(s.srcBuf[s.srcPos]), 1
222
223         if ch >= utf8.RuneSelf {
224             // uncommon case: not ASCII or not enough by
225             for s.srcPos+utf8.UTFMax > s.srcEnd && !utf8
226                 // not enough bytes: read some more,
227                 // save away token text if any
228                 if s.tokPos >= 0 {
229                     s.tokBuf.Write(s.srcBuf[s.to
230                         s.tokPos = 0
231                         // s.tokEnd is set by Scan()
232                 }
233                 // move unread bytes to beginning of
234                 copy(s.srcBuf[0:], s.srcBuf[s.srcPos
235                 s.srcBufOffset += s.srcPos
236                 // read more bytes
237                 // (an io.Reader must return io.EOF
238                 // the end of what it is reading - s
239                 // n == 0 will make this loop retry

```

```

240         // error is in the reader implementa
241         i := s.srcEnd - s.srcPos
242         n, err := s.src.Read(s.srcBuf[i:bufL
243         s.srcPos = 0
244         s.srcEnd = i + n
245         s.srcBuf[s.srcEnd] = utf8.RuneSelf /
246         if err != nil {
247             if s.srcEnd == 0 {
248                 if s.lastCharLen > 0
249                     // previous
250                     s.column++
251             }
252             s.lastCharLen = 0
253             return EOF
254         }
255         if err != io.EOF {
256             s.error(err.Error())
257         }
258         // If err == EOF, we won't b
259         // bytes; break to avoid inf
260         // err is something else, we
261         // we can get more bytes; th
262         break
263     }
264 }
265 // at least one byte
266 ch = rune(s.srcBuf[s.srcPos])
267 if ch >= utf8.RuneSelf {
268     // uncommon case: not ASCII
269     ch, width = utf8.DecodeRune(s.srcBuf
270     if ch == utf8.RuneError && width ==
271         // advance for correct error
272         s.srcPos += width
273         s.lastCharLen = width
274         s.column++
275         s.error("illegal UTF-8 encod
276         return ch
277     }
278 }
279 }
280
281 // advance
282 s.srcPos += width
283 s.lastCharLen = width
284 s.column++
285
286 // special situations
287 switch ch {
288 case 0:

```

```

289         // for compatibility with other tools
290         s.error("illegal character NUL")
291     case '\n':
292         s.line++
293         s.lastLineLen = s.column
294         s.column = 0
295     }
296
297     return ch
298 }
299
300 // Next reads and returns the next Unicode character.
301 // It returns EOF at the end of the source. It reports
302 // a read error by calling s.Error, if not nil; otherwise
303 // it prints an error message to os.Stderr. Next does not
304 // update the Scanner's Position field; use Pos() to
305 // get the current position.
306 func (s *Scanner) Next() rune {
307     s.tokPos = -1 // don't collect token text
308     s.Line = 0    // invalidate token position
309     ch := s.Peek()
310     s.ch = s.next()
311     return ch
312 }
313
314 // Peek returns the next Unicode character in the source with
315 // the scanner. It returns EOF if the scanner's position is
316 // character of the source.
317 func (s *Scanner) Peek() rune {
318     if s.ch < 0 {
319         s.ch = s.next()
320     }
321     return s.ch
322 }
323
324 func (s *Scanner) error(msg string) {
325     s.ErrorCount++
326     if s.Error != nil {
327         s.Error(s, msg)
328         return
329     }
330     pos := s.Position
331     if !pos.IsValid() {
332         pos = s.Pos()
333     }
334     fmt.Fprintf(os.Stderr, "%s: %s\n", pos, msg)
335 }
336
337 func (s *Scanner) scanIdentifier() rune {

```

```

338         ch := s.next() // read character after first '_' or
339         for ch == '_' || unicode.IsLetter(ch) || unicode.IsD
340             ch = s.next()
341     }
342     return ch
343 }
344
345 func digitVal(ch rune) int {
346     switch {
347     case '0' <= ch && ch <= '9':
348         return int(ch - '0')
349     case 'a' <= ch && ch <= 'f':
350         return int(ch - 'a' + 10)
351     case 'A' <= ch && ch <= 'F':
352         return int(ch - 'A' + 10)
353     }
354     return 16 // larger than any legal digit val
355 }
356
357 func isDecimal(ch rune) bool { return '0' <= ch && ch <= '9'
358 }
359 func (s *Scanner) scanMantissa(ch rune) rune {
360     for isDecimal(ch) {
361         ch = s.next()
362     }
363     return ch
364 }
365
366 func (s *Scanner) scanFraction(ch rune) rune {
367     if ch == '.' {
368         ch = s.scanMantissa(s.next())
369     }
370     return ch
371 }
372
373 func (s *Scanner) scanExponent(ch rune) rune {
374     if ch == 'e' || ch == 'E' {
375         ch = s.next()
376         if ch == '-' || ch == '+' {
377             ch = s.next()
378         }
379         ch = s.scanMantissa(ch)
380     }
381     return ch
382 }
383
384 func (s *Scanner) scanNumber(ch rune) (rune, rune) {
385     // isDecimal(ch)
386     if ch == '0' {
387         // int or float

```

```

388         ch = s.next()
389         if ch == 'x' || ch == 'X' {
390             // hexadecimal int
391             ch = s.next()
392             for digitVal(ch) < 16 {
393                 ch = s.next()
394             }
395         } else {
396             // octal int or float
397             seenDecimalDigit := false
398             for isDecimal(ch) {
399                 if ch > '7' {
400                     seenDecimalDigit = t
401                 }
402                 ch = s.next()
403             }
404             if s.Mode&ScanFloats != 0 && (ch ==
405                 // float
406                 ch = s.scanFraction(ch)
407                 ch = s.scanExponent(ch)
408                 return Float, ch
409             }
410             // octal int
411             if seenDecimalDigit {
412                 s.error("illegal octal numbe
413             }
414         }
415         return Int, ch
416     }
417     // decimal int or float
418     ch = s.scanMantissa(ch)
419     if s.Mode&ScanFloats != 0 && (ch == '.' || ch == 'e'
420         // float
421         ch = s.scanFraction(ch)
422         ch = s.scanExponent(ch)
423         return Float, ch
424     }
425     return Int, ch
426 }
427
428 func (s *Scanner) scanDigits(ch rune, base, n int) rune {
429     for n > 0 && digitVal(ch) < base {
430         ch = s.next()
431         n--
432     }
433     if n > 0 {
434         s.error("illegal char escape")
435     }
436     return ch

```

```

437 }
438
439 func (s *Scanner) scanEscape(quote rune) rune {
440     ch := s.next() // read character after '/'
441     switch ch {
442     case 'a', 'b', 'f', 'n', 'r', 't', 'v', '\\', quote:
443         // nothing to do
444         ch = s.next()
445     case '0', '1', '2', '3', '4', '5', '6', '7':
446         ch = s.scanDigits(ch, 8, 3)
447     case 'x':
448         ch = s.scanDigits(s.next(), 16, 2)
449     case 'u':
450         ch = s.scanDigits(s.next(), 16, 4)
451     case 'U':
452         ch = s.scanDigits(s.next(), 16, 8)
453     default:
454         s.error("illegal char escape")
455     }
456     return ch
457 }
458
459 func (s *Scanner) scanString(quote rune) (n int) {
460     ch := s.next() // read character after quote
461     for ch != quote {
462         if ch == '\n' || ch < 0 {
463             s.error("literal not terminated")
464             return
465         }
466         if ch == '\\' {
467             ch = s.scanEscape(quote)
468         } else {
469             ch = s.next()
470         }
471         n++
472     }
473     return
474 }
475
476 func (s *Scanner) scanRawString() {
477     ch := s.next() // read character after '`'
478     for ch != '`' {
479         if ch < 0 {
480             s.error("literal not terminated")
481             return
482         }
483         ch = s.next()
484     }
485 }

```

```

486
487 func (s *Scanner) scanChar() {
488     if s.scanString('\') != 1 {
489         s.error("illegal char literal")
490     }
491 }
492
493 func (s *Scanner) scanComment(ch rune) rune {
494     // ch == '/' || ch == '*'
495     if ch == '/' {
496         // line comment
497         ch = s.next() // read character after "/"
498         for ch != '\n' && ch >= 0 {
499             ch = s.next()
500         }
501         return ch
502     }
503
504     // general comment
505     ch = s.next() // read character after "/*"
506     for {
507         if ch < 0 {
508             s.error("comment not terminated")
509             break
510         }
511         ch0 := ch
512         ch = s.next()
513         if ch0 == '*' && ch == '/' {
514             ch = s.next()
515             break
516         }
517     }
518     return ch
519 }
520
521 // Scan reads the next token or Unicode character from source
522 // It only recognizes tokens t for which the respective Mode
523 // It returns EOF at the end of the source. It reports scan
524 // token errors) by calling s.Error, if not nil; otherwise i
525 // message to os.Stderr.
526 func (s *Scanner) Scan() rune {
527     ch := s.Peek()
528
529     // reset token text position
530     s.tokPos = -1
531     s.Line = 0
532
533 redo:
534     // skip white space
535     for s.Whitespace&(1<<uint(ch)) != 0 {

```

```

536         ch = s.next()
537     }
538
539     // start collecting token text
540     s.tokBuf.Reset()
541     s.tokPos = s.srcPos - s.lastCharLen
542
543     // set token position
544     // (this is a slightly optimized version of the code
545     s.Offset = s.srcBufOffset + s.tokPos
546     if s.column > 0 {
547         // common case: last character was not a '\n'
548         s.Line = s.line
549         s.Column = s.column
550     } else {
551         // last character was a '\n'
552         // (we cannot be at the beginning of the sou
553         // since we have called next() at least once
554         s.Line = s.line - 1
555         s.Column = s.lastLineLen
556     }
557
558     // determine token value
559     tok := ch
560     switch {
561     case unicode.IsLetter(ch) || ch == '_':
562         if s.Mode&ScanIdents != 0 {
563             tok = Ident
564             ch = s.scanIdentifier()
565         } else {
566             ch = s.next()
567         }
568     case isDecimal(ch):
569         if s.Mode&(ScanInts|ScanFloats) != 0 {
570             tok, ch = s.scanNumber(ch)
571         } else {
572             ch = s.next()
573         }
574     default:
575         switch ch {
576         case '"':
577             if s.Mode&ScanStrings != 0 {
578                 s.scanString('"')
579                 tok = String
580             }
581             ch = s.next()
582         case '\\':
583             if s.Mode&ScanChars != 0 {
584                 s.scanChar()

```

```

585             tok = Char
586         }
587         ch = s.next()
588     case '.':
589         ch = s.next()
590         if isDecimal(ch) && s.Mode&ScanFloat
591             tok = Float
592             ch = s.scanMantissa(ch)
593             ch = s.scanExponent(ch)
594     }
595     case '/':
596         ch = s.next()
597         if (ch == '/' || ch == '*') && s.Mod
598             if s.Mode&SkipComments != 0
599                 s.tokPos = -1 // don
600                 ch = s.scanComment(c
601                 goto redo
602             }
603             ch = s.scanComment(ch)
604             tok = Comment
605     }
606     case '`':
607         if s.Mode&ScanRawStrings != 0 {
608             s.scanRawString()
609             tok = String
610         }
611         ch = s.next()
612     default:
613         ch = s.next()
614     }
615 }
616
617 // end of token text
618 s.tokEnd = s.srcPos - s.lastCharLen
619
620 s.ch = ch
621 return tok
622 }
623
624 // Pos returns the position of the character immediately aft
625 // the character or token returned by the last call to Next
626 func (s *Scanner) Pos() (pos Position) {
627     pos.FileName = s.FileName
628     pos.Offset = s.srcBufOffset + s.srcPos - s.lastCharL
629     switch {
630     case s.column > 0:
631         // common case: last character was not a '\n
632         pos.Line = s.line
633         pos.Column = s.column

```

```

634         case s.lastLineLen > 0:
635             // last character was a '\n'
636             pos.Line = s.line - 1
637             pos.Column = s.lastLineLen
638         default:
639             // at the beginning of the source
640             pos.Line = 1
641             pos.Column = 1
642     }
643     return
644 }
645
646 // TokenText returns the string corresponding to the most re
647 // Valid after calling Scan().
648 func (s *Scanner) TokenText() string {
649     if s.tokPos < 0 {
650         // no token text
651         return ""
652     }
653
654     if s.tokEnd < 0 {
655         // if EOF was reached, s.tokEnd is set to -1
656         s.tokEnd = s.tokPos
657     }
658
659     if s.tokBuf.Len() == 0 {
660         // common case: the entire token text is sti
661         return string(s.srcBuf[s.tokPos:s.tokEnd])
662     }
663
664     // part of the token text was saved in tokBuf: save
665     // tokBuf as well and return its content
666     s.tokBuf.Write(s.srcBuf[s.tokPos:s.tokEnd])
667     s.tokPos = s.tokEnd // ensure idempotency of TokenTe
668     return s.tokBuf.String()
669 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/tabwriter/tabwriter.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package tabwriter implements a write filter (tabwriter.Wr
6 // translates tabbed columns in input into properly aligned
7 //
8 // The package is using the Elastic Tabstops algorithm descr
9 // http://nickgravgaard.com/elastic-tabstops/index.html.
10 //
11 package tabwriter
12
13 import (
14     "bytes"
15     "io"
16     "unicode/utf8"
17 )
18
19 // -----
20 // Filter implementation
21
22 // A cell represents a segment of text terminated by tabs or
23 // The text itself is stored in a separate buffer; cell only
24 // segment's size in bytes, its width in runes, and whether
25 // ('\t') terminated cell.
26 //
27 type cell struct {
28     size int // cell size in bytes
29     width int // cell width in runes
30     htab bool // true if the cell is terminated by an h
31 }
32
33 // A Writer is a filter that inserts padding around tab-deli
34 // columns in its input to align them in the output.
35 //
36 // The Writer treats incoming bytes as UTF-8 encoded text co
37 // of cells terminated by (horizontal or vertical) tabs or l
38 // breaks (newline or formfeed characters). Cells in adjacen
39 // constitute a column. The Writer inserts padding as needed
40 // make all cells in a column have the same width, effective
41 // aligning the columns. It assumes that all characters have
```

```

42 // same width except for tabs for which a tabwidth must be s
43 // Note that cells are tab-terminated, not tab-separated: tr
44 // non-tab text at the end of a line does not form a column
45 //
46 // The Writer assumes that all Unicode code points have the
47 // this may not be true in some fonts.
48 //
49 // If DiscardEmptyColumns is set, empty columns that are ter
50 // entirely by vertical (or "soft") tabs are discarded. Colu
51 // terminated by horizontal (or "hard") tabs are not affecte
52 // this flag.
53 //
54 // If a Writer is configured to filter HTML, HTML tags and e
55 // are passed through. The widths of tags and entities are
56 // assumed to be zero (tags) and one (entities) for formatti
57 //
58 // A segment of text may be escaped by bracketing it with Es
59 // characters. The tabwriter passes escaped text segments th
60 // unchanged. In particular, it does not interpret any tabs
61 // breaks within the segment. If the StripEscape flag is set
62 // Escape characters are stripped from the output; otherwise
63 // are passed through as well. For the purpose of formatting
64 // width of the escaped text is always computed excluding th
65 // characters.
66 //
67 // The formfeed character ('\f') acts like a newline but it
68 // terminates all columns in the current line (effectively c
69 // Flush). Cells in the next line start new columns. Unless
70 // inside an HTML tag or inside an escaped text segment, for
71 // characters appear as newlines in the output.
72 //
73 // The Writer must buffer input internally, because proper s
74 // of one line may depend on the cells in future lines. Clie
75 // call Flush when done calling Write.
76 //
77 type Writer struct {
78     // configuration
79     output    io.Writer
80     minwidth  int
81     tabwidth  int
82     padding   int
83     padbytes  [8]byte
84     flags     uint
85
86     // current state
87     buf       bytes.Buffer // collected text excluding tab
88     pos       int           // buffer position up to which
89     cell      cell         // current incomplete cell; cel
90     endChar   byte         // terminating char of escaped
91     lines     [][]cell     // list of lines; each line is

```

```

92         widths []int          // list of column widths in run
93     }
94
95     func (b *Writer) addLine() { b.lines = append(b.lines, []cel
96
97     // Reset the current state.
98     func (b *Writer) reset() {
99         b.buf.Reset()
100        b.pos = 0
101        b.cell = cell{}
102        b.endChar = 0
103        b.lines = b.lines[0:0]
104        b.widths = b.widths[0:0]
105        b.addLine()
106    }
107
108    // Internal representation (current state):
109    //
110    // - all text written is appended to buf; tabs and line brea
111    // - at any given time there is a (possibly empty) incomple
112    //   (the cell starts after a tab or line break)
113    // - cell.size is the number of bytes belonging to the cell
114    // - cell.width is text width in runes of that cell from the
115    //   position pos; html tags and entities are excluded from
116    //   filtering is enabled
117    // - the sizes and widths of processed text are kept in the
118    //   which contains a list of cells for each line
119    // - the widths list is a temporary list with current widths
120    //   formatting; it is kept in Writer because it's re-used
121    //
122    //           |<----- size ----->|
123    //           |                         |
124    //           |<- width ->|<- ignored ->| |
125    //           |                         | |
126    // [---processed---tab-----<tag>...</tag>...]
127    // ^                   ^                         ^
128    // |                   |                         |
129    // buf                 start of incomplete cell pos
130
131    // Formatting can be controlled with these flags.
132    const (
133        // Ignore html tags and treat entities (starting wit
134        // and ending in ';') as single characters (width =
135        FilterHTML uint = 1 << iota
136
137        // Strip Escape characters bracketing escaped text s
138        // instead of passing them through unchanged with th
139        StripEscape
140

```

```

141 // Force right-alignment of cell content.
142 // Default is left-alignment.
143 AlignRight
144
145 // Handle empty columns as if they were not present
146 // the input in the first place.
147 DiscardEmptyColumns
148
149 // Always use tabs for indentation columns (i.e., pa
150 // leading empty cells on the left) independent of p
151 TabIndent
152
153 // Print a vertical bar ('|') between columns (after
154 // Discarded columns appear as zero-width columns ("
155 Debug
156 )
157
158 // A Writer must be initialized with a call to Init. The fir
159 // specifies the filter output. The remaining parameters con
160 //
161 //      minwidth      minimal cell width including any pad
162 //      tabwidth      width of tab characters (equivalent
163 //      padding       padding added to a cell before compu
164 //      padchar       ASCII char used for padding
165 //                  if padchar == '\t', the Writer will
166 //                  width of a '\t' in the formatted out
167 //                  and cells are left-aligned independe
168 //                  (for correct-looking results, tabwid
169 //                  to the tab width in the viewer displ
170 //      flags         formatting control
171 //
172 func (b *Writer) Init(output io.Writer, minwidth, tabwidth,
173     if minwidth < 0 || tabwidth < 0 || padding < 0 {
174         panic("negative minwidth, tabwidth, or paddi
175     }
176     b.output = output
177     b.minwidth = minwidth
178     b.tabwidth = tabwidth
179     b.padding = padding
180     for i := range b.padbytes {
181         b.padbytes[i] = padchar
182     }
183     if padchar == '\t' {
184         // tab padding enforces left-alignment
185         flags &^= AlignRight
186     }
187     b.flags = flags
188
189     b.reset()

```

```

190
191     return b
192 }
193
194 // debugging support (keep code around)
195 func (b *Writer) dump() {
196     pos := 0
197     for i, line := range b.lines {
198         print("(", i, ") ")
199         for _, c := range line {
200             print("[", string(b.buf.Bytes())[pos:
201                 pos += c.size
202         }
203         print("\n")
204     }
205     print("\n")
206 }
207
208 // local error wrapper so we can distinguish errors we want
209 // as errors from genuine panics (which we don't want to ret
210 type osError struct {
211     err error
212 }
213
214 func (b *Writer) write0(buf []byte) {
215     n, err := b.output.Write(buf)
216     if n != len(buf) && err == nil {
217         err = io.ErrShortWrite
218     }
219     if err != nil {
220         panic(osError{err})
221     }
222 }
223
224 func (b *Writer) writeN(src []byte, n int) {
225     for n > len(src) {
226         b.write0(src)
227         n -= len(src)
228     }
229     b.write0(src[0:n])
230 }
231
232 var (
233     newline = []byte{'\n'}
234     tabs    = []byte{"\t\t\t\t\t\t\t\t\t\t"}
235 )
236
237 func (b *Writer) writePadding(textw, cellw int, useTabs bool
238     if b.padbytes[0] == '\t' || useTabs {
239         // padding is done with tabs

```

```

240         if b.tabwidth == 0 {
241             return // tabs have no width - can't
242         }
243         // make cellw the smallest multiple of b.tab
244         cellw = (cellw + b.tabwidth - 1) / b.tabwidth
245         n := cellw - textw // amount of padding
246         if n < 0 {
247             panic("internal error")
248         }
249         b.writeN(tabs, (n+b.tabwidth-1)/b.tabwidth)
250         return
251     }
252
253     // padding is done with non-tab characters
254     b.writeN(b.padbytes[0:], cellw-textw)
255 }
256
257 var vbar = []byte{'|'}
258
259 func (b *Writer) writeLines(pos0 int, line0, line1 int) (pos
260     pos = pos0
261     for i := line0; i < line1; i++ {
262         line := b.lines[i]
263
264         // if TabIndent is set, use tabs to pad lead
265         useTabs := b.flags&TabIndent != 0
266
267         for j, c := range line {
268             if j > 0 && b.flags&Debug != 0 {
269                 // indicate column break
270                 b.write0(vbar)
271             }
272
273             if c.size == 0 {
274                 // empty cell
275                 if j < len(b.widths) {
276                     b.writePadding(c.wid
277                 }
278             } else {
279                 // non-empty cell
280                 useTabs = false
281                 if b.flags&AlignRight == 0 {
282                     b.write0(b.buf.Bytes
283                     pos += c.size
284                     if j < len(b.widths)
285                         b.writePaddi
286                 }
287             } else { // align right
288                 if j < len(b.widths)

```

```

289                                     b.writePaddi
290                                     }
291                                     b.write0(b.buf.Bytes
292                                     pos += c.size
293                                     }
294                                 }
295                             }
296
297                             if i+1 == len(b.lines) {
298                                 // last buffered line - we don't hav
299                                 // any outstanding buffered data
300                                 b.write0(b.buf.Bytes())[pos : pos+b.c
301                                 pos += b.cell.size
302                             } else {
303                                 // not the last line - write newline
304                                 b.write0(newline)
305                             }
306                         }
307                         return
308 }
309
310 // Format the text between line0 and line1 (excluding line1)
311 // is the buffer position corresponding to the beginning of
312 // Returns the buffer position corresponding to the beginnin
313 // line1 and an error, if any.
314 //
315 func (b *Writer) format(pos0 int, line0, line1 int) (pos int
316     pos = pos0
317     column := len(b.widths)
318     for this := line0; this < line1; this++ {
319         line := b.lines[this]
320
321         if column < len(line)-1 {
322             // cell exists in this column => thi
323             // has more cells than the previous
324             // (the last cell per line is ignore
325             // tab-terminated; the last cell per
326             // text before the newline/formfeed
327             // to a column)
328
329             // print unprinted lines until begin
330             pos = b.writeLines(pos, line0, this)
331             line0 = this
332
333             // column block begin
334             width := b.minwidth // minimal colum
335             discardable := true // true if all c
336             for ; this < line1; this++ {
337                 line = b.lines[this]

```

```

338         if column < len(line)-1 {
339             // cell exists in th
340             c := line[column]
341             // update width
342             if w := c.width + b.
343                 width = w
344             }
345             // update discardabl
346             if c.width > 0 || c.
347                 discardable
348             }
349         } else {
350             break
351         }
352     }
353     // column block end
354
355     // discard empty columns if necessar
356     if discardable && b.flags&DiscardEmp
357         width = 0
358     }
359
360     // format and print all columns to t
361     // (we know the widths of this colum
362     b.widths = append(b.widths, width) /
363     pos = b.format(pos, line0, this)
364     b.widths = b.widths[0 : len(b.widths
365     line0 = this
366     }
367 }
368
369 // print unprinted lines until end
370 return b.writelines(pos, line0, line1)
371 }
372
373 // Append text to current cell.
374 func (b *Writer) append(text []byte) {
375     b.buf.Write(text)
376     b.cell.size += len(text)
377 }
378
379 // Update the cell width.
380 func (b *Writer) updateWidth() {
381     b.cell.width += utf8.RuneCount(b.buf.Bytes())[b.pos:b
382     b.pos = b.buf.Len()
383 }
384
385 // To escape a text segment, bracket it with Escape characte
386 // For instance, the tab in this string "Ignore this tab: \x
387 // does not terminate a cell and constitutes a single charac

```

```

388 // width one for formatting purposes.
389 //
390 // The value 0xff was chosen because it cannot appear in a v
391 //
392 const Escape = '\xff'
393
394 // Start escaped mode.
395 func (b *Writer) startEscape(ch byte) {
396     switch ch {
397     case Escape:
398         b.endChar = Escape
399     case '<':
400         b.endChar = '>'
401     case '&':
402         b.endChar = ';'
403     }
404 }
405
406 // Terminate escaped mode. If the escaped text was an HTML t
407 // is assumed to be zero for formatting purposes; if it was
408 // its width is assumed to be one. In all other cases, the w
409 // unicode width of the text.
410 //
411 func (b *Writer) endEscape() {
412     switch b.endChar {
413     case Escape:
414         b.updateWidth()
415         if b.flags&StripEscape == 0 {
416             b.cell.width -= 2 // don't count the
417         }
418     case '>': // tag of zero width
419     case ';':
420         b.cell.width++ // entity, count as one rune
421     }
422     b.pos = b.buf.Len()
423     b.endChar = 0
424 }
425
426 // Terminate the current cell by adding it to the list of ce
427 // current line. Returns the number of cells in that line.
428 //
429 func (b *Writer) terminateCell(htab bool) int {
430     b.cell.htab = htab
431     line := &b.lines[len(b.lines)-1]
432     *line = append(*line, b.cell)
433     b.cell = cell{}
434     return len(*line)
435 }
436

```

```

437 func handlePanic(err *error) {
438     if e := recover(); e != nil {
439         *err = e.(osError).err // re-panics if it's
440     }
441 }
442
443 // Flush should be called after the last call to Write to en
444 // that any data buffered in the Writer is written to output
445 // incomplete escape sequence at the end is considered
446 // complete for formatting purposes.
447 //
448 func (b *writer) Flush() (err error) {
449     defer b.reset() // even in the presence of errors
450     defer handlePanic(&err)
451
452     // add current cell if not empty
453     if b.cell.size > 0 {
454         if b.endChar != 0 {
455             // inside escape - terminate it even
456             b.endEscape()
457         }
458         b.terminateCell(false)
459     }
460
461     // format contents of buffer
462     b.format(0, 0, len(b.lines))
463
464     return
465 }
466
467 var hbar = []byte("---\n")
468
469 // Write writes buf to the writer b.
470 // The only errors returned are ones encountered
471 // while writing to the underlying output stream.
472 //
473 func (b *writer) Write(buf []byte) (n int, err error) {
474     defer handlePanic(&err)
475
476     // split text into cells
477     n = 0
478     for i, ch := range buf {
479         if b.endChar == 0 {
480             // outside escape
481             switch ch {
482             case '\t', '\v', '\n', '\f':
483                 // end of cell
484                 b.append(buf[n:i])
485                 b.updateWidth()

```

```

486         n = i + 1 // ch consumed
487         ncells := b.terminateCell(ch
488         if ch == '\n' || ch == '\f'
489             // terminate line
490             b.addLine()
491             if ch == '\f' || nce
492                 // A '\f' al
493                 // line has
494                 // the forma
495                 // line is i
496                 // Writer co
497                 if err = b.F
498                     retu
499             }
500             if ch == '\f
501                 // i
502                 b.wr
503             }
504         }
505     }
506
507     case Escape:
508         // start of escaped sequence
509         b.append(buf[n:i])
510         b.updateWidth()
511         n = i
512         if b.flags&StripEscape != 0
513             n++ // strip Escape
514         }
515         b.startEscape(Escape)
516
517     case '<', '&':
518         // possibly an html tag/enti
519         if b.flags&FilterHTML != 0 {
520             // begin of tag/enti
521             b.append(buf[n:i])
522             b.updateWidth()
523             n = i
524             b.startEscape(ch)
525         }
526     }
527
528 } else {
529     // inside escape
530     if ch == b.endChar {
531         // end of tag/entity
532         j := i + 1
533         if ch == Escape && b.flags&S
534             j = i // strip Escap
535     }

```

```

536             b.append(buf[n:j])
537             n = i + 1 // ch consumed
538             b.endEscape()
539         }
540     }
541 }
542
543 // append leftover text
544 b.append(buf[n:])
545 n = len(buf)
546 return
547 }
548
549 // NewWriter allocates and initializes a new tabwriter.Write
550 // The parameters are the same as for the the Init function.
551 //
552 func NewWriter(output io.Writer, minwidth, tabwidth, padding
553             return new(Writer).Init(output, minwidth, tabwidth,
554 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/template/doc.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6 Package template implements data-driven templates for genera
7
8 To generate HTML output, see package html/template, which ha
9 as this package but automatically secures HTML output agains
10
11 Templates are executed by applying them to a data structure.
12 template refer to elements of the data structure (typically
13 or a key in a map) to control execution and derive values to
14 Execution of the template walks the structure and sets the c
15 by a period '.' and called "dot", to the value at the curren
16 structure as execution proceeds.
17
18 The input text for a template is UTF-8-encoded text in any f
19 "Actions"--data evaluations or control structures--are delir
20 "{{" and "}}"; all text outside actions is copied to the out
21 Actions may not span newlines, although comments can.
22
23 Once constructed, a template may be executed safely in paral
24
25 Here is a trivial example that prints "17 items are made of
26
27         type Inventory struct {
28             Material string
29             Count    uint
30         }
31         sweaters := Inventory{"wool", 17}
32         tmpl, err := template.New("test").Parse("{{.Count}}
33         if err != nil { panic(err) }
34         err = tmpl.Execute(os.Stdout, sweaters)
35         if err != nil { panic(err) }
36
37 More intricate examples appear below.
38
39 Actions
40
41 Here is the list of actions. "Arguments" and "pipelines" are
```

```

42 data, defined in detail below.
43
44 */
45 //      {{{/* a comment */}}}
46 //      A comment; discarded. May contain newlines.
47 //      Comments do not nest.
48 /*
49
50      {{{pipeline}}}
51          The default textual representation of the va
52          is copied to the output.
53
54      {{{if pipeline}}} T1 {{{end}}}
55          If the value of the pipeline is empty, no ou
56          otherwise, T1 is executed. The empty values
57          nil pointer or interface value, and any arra
58          string of length zero.
59          Dot is unaffected.
60
61      {{{if pipeline}}} T1 {{{else}}} T0 {{{end}}}
62          If the value of the pipeline is empty, T0 is
63          otherwise, T1 is executed. Dot is unaffecte
64
65      {{{range pipeline}}} T1 {{{end}}}
66          The value of the pipeline must be an array,
67          the value of the pipeline has length zero, n
68          otherwise, dot is set to the successive elem
69          slice, or map and T1 is executed. If the val
70          keys are of basic type with a defined order
71          elements will be visited in sorted key order
72
73      {{{range pipeline}}} T1 {{{else}}} T0 {{{end}}}
74          The value of the pipeline must be an array,
75          the value of the pipeline has length zero, d
76          T0 is executed; otherwise, dot is set to the
77          of the array, slice, or map and T1 is execut
78
79      {{{template "name"}}}
80          The template with the specified name is exec
81
82      {{{template "name" pipeline}}}
83          The template with the specified name is exec
84          to the value of the pipeline.
85
86      {{{with pipeline}}} T1 {{{end}}}
87          If the value of the pipeline is empty, no ou
88          otherwise, dot is set to the value of the pi
89          executed.
90
91      {{{with pipeline}}} T1 {{{else}}} T0 {{{end}}}

```

```

92             If the value of the pipeline is empty, dot i
93             is executed; otherwise, dot is set to the va
94             and T1 is executed.
95
96 Arguments
97
98 An argument is a simple value, denoted by one of the followi
99
100 - A boolean, string, character, integer, floating-po
101   or complex constant in Go syntax. These behave lik
102   constants, although raw strings may not span newli
103 - The character '.' (period):
104   .
105   The result is the value of dot.
106 - A variable name, which is a (possibly empty) alpha
107   preceded by a dollar sign, such as
108     $piOver2
109   or
110     $
111   The result is the value of the variable.
112   Variables are described below.
113 - The name of a field of the data, which must be a s
114   by a period, such as
115     .Field
116   The result is the value of the field. Field invoca
117   chained:
118     .Field1.Field2
119   Fields can also be evaluated on variables, includi
120     $x.Field1.Field2
121 - The name of a key of the data, which must be a map
122   by a period, such as
123     .Key
124   The result is the map element value indexed by the
125   Key invocations may be chained and combined with f
126   depth:
127     .Field1.Key1.Field2.Key2
128   Although the key must be an alphanumeric identifie
129   field names they do not need to start with an uppe
130   Keys can also be evaluated on variables, including
131     $x.key1.key2
132 - The name of a niladic method of the data, preceded
133   such as
134     .Method
135   The result is the value of invoking the method wit
136   receiver, dot.Method(). Such a method must have on
137   any type) or two return values, the second of whic
138   If it has two and the returned error is non-nil, e
139   and an error is returned to the caller as the valu
140   Method invocations may be chained and combined wit

```

141 to any depth:
142 .Field1.Key1.Method1.Field2.Key2.Method2
143 Methods can also be evaluated on variables, includ
144 \$x.Method1.Field
145 - The name of a niladic function, such as
146 fun
147 The result is the value of invoking the function,
148 types and values behave as in methods. Functions a
149 names are described below.
150
151 Arguments may evaluate to any type; if they are pointers the
152 automatically indirects to the base type when required.
153 If an evaluation yields a function value, such as a function
154 field of a struct, the function is not invoked automatically
155 can be used as a truth value for an if action and the like.
156 it, use the call function, defined below.
157
158 A pipeline is a possibly chained sequence of "commands". A c
159 value (argument) or a function or method call, possibly with
160
161 Argument
162 The result is the value of evaluating the ar
163 .Method [Argument...]
164 The method can be alone or the last element
165 unlike methods in the middle of a chain, it
166 The result is the value of calling the metho
167 arguments:
168 dot.Method(Argument1, etc.)
169 functionName [Argument...]
170 The result is the value of calling the funct
171 with the name:
172 function(Argument1, etc.)
173 Functions and function names are described b
174
175 Pipelines
176
177 A pipeline may be "chained" by separating a sequence of comm
178 characters '|'. In a chained pipeline, the result of the eac
179 passed as the last argument of the following command. The ou
180 command in the pipeline is the value of the pipeline.
181
182 The output of a command will be either one value or two valu
183 which has type error. If that second value is present and ev
184 non-nil, execution terminates and the error is returned to t
185 Execute.
186
187 Variables
188
189 A pipeline inside an action may initialize a variable to cap

190 The initialization has syntax
191
192 \$variable := pipeline
193
194 where \$variable is the name of the variable. An action that
195 variable produces no output.
196
197 If a "range" action initializes a variable, the variable is
198 successive elements of the iteration. Also, a "range" may d
199 variables, separated by a comma:
200
201 \$index, \$element := pipeline
202
203 in which case \$index and \$element are set to the successive
204 array/slice index or map key and element, respectively. Not
205 only one variable, it is assigned the element; this is oppos
206 convention in Go range clauses.
207
208 A variable's scope extends to the "end" action of the contro
209 "with", or "range") in which it is declared, or to the end o
210 there is no such control structure. A template invocation d
211 variables from the point of its invocation.
212
213 When execution begins, \$ is set to the data argument passed
214 to the starting value of dot.
215
216 Examples
217
218 Here are some example one-line templates demonstrating pipel
219 All produce the quoted word "output":
220
221 {{"\\"output\\"}}
222 A string constant.
223 {{`"output"`}}
224 A raw string constant.
225 {{printf "%q" "output"}}
226 A function call.
227 {{"output" | printf "%q"}}
228 A function call whose final argument comes f
229 command.
230 {{"put" | printf "%s%s" "out" | printf "%q"}}
231 A more elaborate call.
232 {{"output" | printf "%s" | printf "%q"}}
233 A longer chain.
234 {{with "output"}}{{printf "%q" .}}{{end}}
235 A with action using dot.
236 {{with \$x := "output" | printf "%q"}}{{\$x}}{{end}}
237 A with action that creates and uses a variab
238 {{with \$x := "output"}}{{printf "%q" \$x}}{{end}}
239 A with action that uses the variable in anot

```
240         {{with $x := "output"}}{{ $x | printf "%q" }}{{end}}
241         The same, but pipelined.
242
243 Functions
244
245 During execution functions are found in two function maps: f
246 template, then in the global function map. By default, no fu
247 in the template but the Funcs method can be used to add them
248
249 Predefined global functions are named as follows.
250
251     and
252         Returns the boolean AND of its arguments by
253         first empty argument or the last argument, t
254         "and x y" behaves as "if x then y else x". A
255         arguments are evaluated.
256
257     call
258         Returns the result of calling the first argu
259         must be a function, with the remaining argum
260         Thus "call .X.Y 1 2" is, in Go notation, dot
261         Y is a func-valued field, map entry, or the
262         The first argument must be the result of an
263         that yields a value of function type (as dis
264         a predefined function such as print). The fu
265         return either one or two result values, the
266         is of type error. If the arguments don't mat
267         or the returned error value is non-nil, exec
268
269     html
270         Returns the escaped HTML equivalent of the t
271         representation of its arguments.
272
273     index
274         Returns the result of indexing its first arg
275         following arguments. Thus "index x 1 2 3" is
276         x[1][2][3]. Each indexed item must be a map,
277
278     js
279         Returns the escaped JavaScript equivalent of
280         representation of its arguments.
281
282     len
283         Returns the integer length of its argument.
284
285     not
286         Returns the boolean negation of its single a
287
288     or
289         Returns the boolean OR of its arguments by r
290         first non-empty argument or the last argumen
291         "or x y" behaves as "if x then x else y". Al
292         arguments are evaluated.
293
294     print
295         An alias for fmt.Sprint
296
297     printf
```

289 An alias for fmt.Sprintf
 290 println
 291 An alias for fmt.Sprintln
 292 urlquery
 293 Returns the escaped value of the textual rep
 294 its arguments in a form suitable for embeddi
 295
 296 The boolean functions take any zero value to be false and a
 297 be true.
 298
 299 Associated templates
 300
 301 Each template is named by a string specified when it is crea
 302 template is associated with zero or more other templates tha
 303 name; such associations are transitive and form a name space
 304
 305 A template may use a template invocation to instantiate anot
 306 template; see the explanation of the "template" action above
 307 that of a template associated with the template that contain
 308
 309 Nested template definitions
 310
 311 When parsing a template, another template may be defined and
 312 template being parsed. Template definitions must appear at t
 313 template, much like global variables in a Go program.
 314
 315 The syntax of such definitions is to surround each template
 316 "define" and "end" action.
 317
 318 The define action names the template being created by provid
 319 constant. Here is a simple example:
 320
 321 `{{define "T1"}}ONE{{end}}
 322 {{define "T2"}}TWO{{end}}
 323 {{define "T3"}}{{template "T1"}} {{template "T2"}}{{
 324 {{template "T3"}}`
 325
 326 This defines two templates, T1 and T2, and a third T3 that i
 327 when it is executed. Finally it invokes T3. If executed this
 328 produce the text
 329
 330 ONE TWO
 331
 332 By construction, a template may reside in only one associati
 333 necessary to have a template addressable from multiple assoc
 334 template definition must be parsed multiple times to create
 335 values, or must be copied with the Clone or AddParseTree met
 336
 337 Parse may be called multiple times to assemble the various a

```
338 see the ParseFiles and ParseGlob functions and methods for s
339 related templates stored in files.
340
341 A template may be executed directly or through ExecuteTempla
342 an associated template identified by name. To invoke our exa
343 might write,
344
345     err := tmpl.Execute(os.Stdout, "no data needed")
346     if err != nil {
347         log.Fatalf("execution failed: %s", err)
348     }
349
350 or to invoke a particular template explicitly by name,
351
352     err := tmpl.ExecuteTemplate(os.Stdout, "T2", "no dat
353     if err != nil {
354         log.Fatalf("execution failed: %s", err)
355     }
356
357 */
358 package template
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/template/exec.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "fmt"
9     "io"
10    "reflect"
11    "runtime"
12    "sort"
13    "strings"
14    "text/template/parse"
15 )
16
17 // state represents the state of an execution. It's not part
18 // template so that multiple executions of the same template
19 // can execute in parallel.
20 type state struct {
21     tmpl *Template
22     wr    io.Writer
23     line int        // line number for errors
24     vars []variable // push-down stack of variable value
25 }
26
27 // variable holds the dynamic value of a variable such as $,
28 type variable struct {
29     name string
30     value reflect.Value
31 }
32
33 // push pushes a new variable on the stack.
34 func (s *state) push(name string, value reflect.Value) {
35     s.vars = append(s.vars, variable{name, value})
36 }
37
38 // mark returns the length of the variable stack.
39 func (s *state) mark() int {
40     return len(s.vars)
41 }
```

```

42
43 // pop pops the variable stack up to the mark.
44 func (s *state) pop(mark int) {
45     s.vars = s.vars[0:mark]
46 }
47
48 // setVar overwrites the top-nth variable on the stack. Used
49 func (s *state) setVar(n int, value reflect.Value) {
50     s.vars[len(s.vars)-n].value = value
51 }
52
53 // varValue returns the value of the named variable.
54 func (s *state) varValue(name string) reflect.Value {
55     for i := s.mark() - 1; i >= 0; i-- {
56         if s.vars[i].name == name {
57             return s.vars[i].value
58         }
59     }
60     s.errorf("undefined variable: %s", name)
61     return zero
62 }
63
64 var zero reflect.Value
65
66 // errorf formats the error and terminates processing.
67 func (s *state) errorf(format string, args ...interface{}) {
68     format = fmt.Sprintf("template: %s:%d: %s", s.tmpl.N
69     panic(fmt.Errorf(format, args...))
70 }
71
72 // error terminates processing.
73 func (s *state) error(err error) {
74     s.errorf("%s", err)
75 }
76
77 // errRecover is the handler that turns panics into returns
78 // level of Parse.
79 func errRecover(errp *error) {
80     e := recover()
81     if e != nil {
82         switch err := e.(type) {
83             case runtime.Error:
84                 panic(e)
85             case error:
86                 *errp = err
87             default:
88                 panic(e)
89         }
90     }
91 }

```

```

92
93 // ExecuteTemplate applies the template associated with t th
94 // to the specified data object and writes the output to wr.
95 func (t *Template) ExecuteTemplate(wr io.Writer, name string
96     tmpl := t.tmpl[name]
97     if tmpl == nil {
98         return fmt.Errorf("template: no template %q
99     }
100     return tmpl.Execute(wr, data)
101 }
102
103 // Execute applies a parsed template to the specified data o
104 // and writes the output to wr.
105 func (t *Template) Execute(wr io.Writer, data interface{}) (
106     defer errRecover(&err)
107     value := reflect.ValueOf(data)
108     state := &state{
109         tmpl: t,
110         wr:    wr,
111         line: 1,
112         vars: []variable{{"$", value}},
113     }
114     if t.Tree == nil || t.Root == nil {
115         state.Errorf("%q is an incomplete or empty t
116     }
117     state.walk(value, t.Root)
118     return
119 }
120
121 // Walk functions step through the major pieces of the templ
122 // generating output as they go.
123 func (s *state) walk(dot reflect.Value, n parse.Node) {
124     switch n := n.(type) {
125     case *parse.ActionNode:
126         s.line = n.Line
127         // Do not pop variables so they persist unti
128         // Also, if the action declares variables, d
129         val := s.evalPipeline(dot, n.Pipe)
130         if len(n.Pipe.Decl) == 0 {
131             s.printValue(n, val)
132         }
133     case *parse.IfNode:
134         s.line = n.Line
135         s.walkIfOrWith(parse.NodeIf, dot, n.Pipe, n.
136     case *parse.ListNode:
137         for _, node := range n.Nodes {
138             s.walk(dot, node)
139         }
140     case *parse.RangeNode:

```

```

141         s.line = n.Line
142         s.walkRange(dot, n)
143     case *parse.TemplateNode:
144         s.line = n.Line
145         s.walkTemplate(dot, n)
146     case *parse.TextNode:
147         if _, err := s.wr.Write(n.Text); err != nil
148             s.error(err)
149     }
150     case *parse.WithNode:
151         s.line = n.Line
152         s.walkIfOrWith(parse.NodeWith, dot, n.Pipe,
153     default:
154         s.errorf("unknown node: %s", n)
155     }
156 }
157
158 // walkIfOrWith walks an 'if' or 'with' node. The two contro
159 // are identical in behavior except that 'with' sets dot.
160 func (s *state) walkIfOrWith(typ parse.NodeType, dot reflect
161     defer s.pop(s.mark())
162     val := s.evalPipeline(dot, pipe)
163     truth, ok := isTrue(val)
164     if !ok {
165         s.errorf("if/with can't use %v", val)
166     }
167     if truth {
168         if typ == parse.NodeWith {
169             s.walk(val, list)
170         } else {
171             s.walk(dot, list)
172         }
173     } else if elseList != nil {
174         s.walk(dot, elseList)
175     }
176 }
177
178 // isTrue returns whether the value is 'true', in the sense
179 // and whether the value has a meaningful truth value.
180 func isTrue(val reflect.Value) (truth, ok bool) {
181     if !val.IsValid() {
182         // Something like var x interface{}, never s
183         return false, true
184     }
185     switch val.Kind() {
186     case reflect.Array, reflect.Map, reflect.Slice, refl
187         truth = val.Len() > 0
188     case reflect.Bool:
189         truth = val.Bool()

```

```

190     case reflect.Complex64, reflect.Complex128:
191         truth = val.Complex() != 0
192     case reflect.Chan, reflect.Func, reflect.Ptr, reflect.ReflectMap, reflect.Slice:
193         truth = !val.IsNil()
194     case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32, reflect.Int64:
195         truth = val.Int() != 0
196     case reflect.Float32, reflect.Float64:
197         truth = val.Float() != 0
198     case reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32, reflect.Uint64:
199         truth = val.Uint() != 0
200     case reflect.Struct:
201         truth = true // Struct values are always true
202     default:
203         return
204     }
205     return truth, true
206 }
207
208 func (s *state) walkRange(dot reflect.Value, r *parse.Ranger) {
209     defer s.pop(s.mark())
210     val, _ := indirect(s.evalPipeline(dot, r.Pipe))
211     // mark top of stack before any variables in the body
212     mark := s.mark()
213     oneIteration := func(index, elem reflect.Value) {
214         // Set top var (lexically the second if there are two)
215         if len(r.Pipe.Decl) > 0 {
216             s.setVar(1, elem)
217         }
218         // Set next var (lexically the first if there are two)
219         if len(r.Pipe.Decl) > 1 {
220             s.setVar(2, index)
221         }
222         s.walk(elem, r.List)
223         s.pop(mark)
224     }
225     switch val.Kind() {
226     case reflect.Array, reflect.Slice:
227         if val.Len() == 0 {
228             break
229         }
230         for i := 0; i < val.Len(); i++ {
231             oneIteration(reflect.ValueOf(i), val.Index(i))
232         }
233         return
234     case reflect.Map:
235         if val.Len() == 0 {
236             break
237         }
238         for _, key := range sortKeys(val.MapKeys()) {
239             oneIteration(key, val.MapIndex(key))

```

```

240     }
241     return
242 case reflect.Chan:
243     if val.IsNil() {
244         break
245     }
246     i := 0
247     for ; ; i++ {
248         elem, ok := val.Recv()
249         if !ok {
250             break
251         }
252         oneIteration(reflect.ValueOf(i), ele
253     }
254     if i == 0 {
255         break
256     }
257     return
258 case reflect.Invalid:
259     break // An invalid value is likely a nil ma
260 default:
261     s.errorf("range can't iterate over %v", val)
262 }
263 if r.ElseList != nil {
264     s.walk(dot, r.ElseList)
265 }
266 }
267
268 func (s *state) walkTemplate(dot reflect.Value, t *parse.Tem
269     tmpl := s.tmpl.tmpl[t.Name]
270     if tmpl == nil {
271         s.errorf("template %q not defined", t.Name)
272     }
273     // Variables declared by the pipeline persist.
274     dot = s.evalPipeline(dot, t.Pipe)
275     newState := *s
276     newState.tmpl = tmpl
277     // No dynamic scoping: template invocations inherit
278     newState.vars = []variable{{"$", dot}}
279     newState.walk(dot, tmpl.Root)
280 }
281
282 // Eval functions evaluate pipelines, commands, and their el
283 // values from the data structure by examining fields, calli
284 // The printing of those values happens only through walk fu
285
286 // evalPipeline returns the value acquired by evaluating a p
287 // pipeline has a variable declaration, the variable will be
288 // stack. Callers should therefore pop the stack after they

```

```

289 // executing commands depending on the pipeline value.
290 func (s *state) evalPipeline(dot reflect.Value, pipe *parse.
291     if pipe == nil {
292         return
293     }
294     for _, cmd := range pipe.Cmds {
295         value = s.evalCommand(dot, cmd, value) // pr
296         // If the object has type interface{}, dig d
297         if value.Kind() == reflect.Interface && valu
298             value = reflect.ValueOf(value.Interf
299     }
300 }
301 for _, variable := range pipe.Decl {
302     s.push(variable.Ident[0], value)
303 }
304 return value
305 }
306
307 func (s *state) notAFunction(args []parse.Node, final reflec
308     if len(args) > 1 || final.IsValid() {
309         s.errorf("can't give argument to non-functio
310     }
311 }
312
313 func (s *state) evalCommand(dot reflect.Value, cmd *parse.Co
314     firstWord := cmd.Args[0]
315     switch n := firstWord.(type) {
316     case *parse.FieldNode:
317         return s.evalFieldNode(dot, n, cmd.Args, fin
318     case *parse.IdentifierNode:
319         // Must be a function.
320         return s.evalFunction(dot, n.Ident, cmd.Args
321     case *parse.VariableNode:
322         return s.evalVariableNode(dot, n, cmd.Args,
323     }
324     s.notAFunction(cmd.Args, final)
325     switch word := firstWord.(type) {
326     case *parse.BoolNode:
327         return reflect.ValueOf(word.True)
328     case *parse.DotNode:
329         return dot
330     case *parse.NumberNode:
331         return s.idealConstant(word)
332     case *parse.StringNode:
333         return reflect.ValueOf(word.Text)
334     }
335     s.errorf("can't evaluate command %q", firstWord)
336     panic("not reached")
337 }

```

```

338
339 // idealConstant is called to return the value of a number i
340 // we don't know the type. In that case, the syntax of the n
341 // its type, and we use Go rules to resolve. Note there is
342 // a uint ideal constant in this situation - the value must
343 func (s *state) idealConstant(constant *parse.NumberNode) re
344     // These are ideal constants but we don't know the t
345     // and we have no context. (If it was a method argu
346     // we'd know what we need.) The syntax guides us to
347     switch {
348     case constant.IsComplex:
349         return reflect.ValueOf(constant.Complex128)
350     case constant.IsFloat && strings.IndexAny(constant.T
351         return reflect.ValueOf(constant.Float64)
352     case constant.IsInt:
353         n := int(constant.Int64)
354         if int64(n) != constant.Int64 {
355             s.errorf("%s overflows int", constan
356         }
357         return reflect.ValueOf(n)
358     case constant.IsUint:
359         s.errorf("%s overflows int", constant.Text)
360     }
361     return zero
362 }
363
364 func (s *state) evalFieldNode(dot reflect.Value, field *pars
365     return s.evalFieldChain(dot, dot, field.Ident, args,
366 }
367
368 func (s *state) evalVariableNode(dot reflect.Value, v *parse
369     // $x.Field has $x as the first ident, Field as the
370     value := s.varValue(v.Ident[0])
371     if len(v.Ident) == 1 {
372         s.notAFunction(args, final)
373         return value
374     }
375     return s.evalFieldChain(dot, value, v.Ident[1:], arg
376 }
377
378 // evalFieldChain evaluates .X.Y.Z possibly followed by argu
379 // dot is the environment in which to evaluate arguments, wh
380 // receiver is the value being walked along the chain.
381 func (s *state) evalFieldChain(dot, receiver reflect.Value,
382     n := len(ident)
383     for i := 0; i < n-1; i++ {
384         receiver = s.evalField(dot, ident[i], nil, z
385     }
386     // Now if it's a method, it gets the arguments.
387     return s.evalField(dot, ident[n-1], args, final, rec

```

```

388 }
389
390 func (s *state) evalFunction(dot reflect.Value, name string,
391     function, ok := findFunction(name, s.tmpl)
392     if !ok {
393         s.errorf("%q is not a defined function", name)
394     }
395     return s.evalCall(dot, function, name, args, final)
396 }
397
398 // evalField evaluates an expression like (.Field) or (.Field)
399 // The 'final' argument represents the return value from the
400 // value of the pipeline, if any.
401 func (s *state) evalField(dot reflect.Value, fieldName string,
402     if !receiver.IsValid() {
403         return zero
404     }
405     typ := receiver.Type()
406     receiver, _ = indirect(receiver)
407     // Unless it's an interface, need to get to a value
408     // we see all methods of T and *T.
409     ptr := receiver
410     if ptr.Kind() != reflect.Interface && ptr.CanAddr()
411         ptr = ptr.Addr()
412     }
413     if method := ptr.MethodByName(fieldName); method.IsValid()
414         return s.evalCall(dot, method, fieldName, args, final)
415     }
416     hasArgs := len(args) > 1 || final.IsValid()
417     // It's not a method; is it a field of a struct?
418     receiver, isNil := indirect(receiver)
419     if receiver.Kind() == reflect.Struct {
420         tField, ok := receiver.Type().FieldByName(fieldName)
421         if ok {
422             field := receiver.FieldByIndex(tField.Index)
423             if tField.PkgPath == "" { // field is exported
424                 // If it's a function, we must call it
425                 if hasArgs {
426                     s.errorf("%s has arguments", fieldName)
427                 }
428                 return field
429             }
430         }
431     }
432     // If it's a map, attempt to use the field name as a key
433     if receiver.Kind() == reflect.Map {
434         nameVal := reflect.ValueOf(fieldName)
435         if nameVal.Type().AssignableTo(receiver.Type())
436             if hasArgs {

```

```

437             s.errorf("%s is not a method
438         }
439         return receiver.MapIndex(nameVal)
440     }
441 }
442 if isNil {
443     s.errorf("nil pointer evaluating %s.%s", typ
444 }
445 s.errorf("can't evaluate field %s in type %s", field
446 panic("not reached")
447 }
448
449 var (
450     errorType      = reflect.TypeOf((*error)(nil)).Elem
451     fmtStringerType = reflect.TypeOf((*fmt.Stringer)(nil)
452 )
453
454 // evalCall executes a function or method call. If it's a me
455 // it looks just like a function call. The arg list, if non
456 // as the function itself.
457 func (s *state) evalCall(dot, fun reflect.Value, name string
458     if args != nil {
459         args = args[1:] // Zeroth arg is function na
460     }
461     typ := fun.Type()
462     numIn := len(args)
463     if final.IsValid() {
464         numIn++
465     }
466     numFixed := len(args)
467     if typ.IsVariadic() {
468         numFixed = typ.NumIn() - 1 // last arg is th
469         if numIn < numFixed {
470             s.errorf("wrong number of args for %
471         }
472     } else if numIn < typ.NumIn()-1 || !typ.IsVariadic()
473         s.errorf("wrong number of args for %s: want
474     }
475     if !goodFunc(typ) {
476         s.errorf("can't handle multiple results from
477     }
478     // Build the arg list.
479     argv := make([]reflect.Value, numIn)
480     // Args must be evaluated. Fixed args first.
481     i := 0
482     for ; i < numFixed; i++ {
483         argv[i] = s.evalArg(dot, typ.In(i), args[i])
484     }
485     // Now the ... args.

```

```

486     if typ.IsVariadic() {
487         argType := typ.In(typ.NumIn() - 1).Elem() //
488         for ; i < len(args); i++ {
489             argv[i] = s.evalArg(dot, argType, ar
490         }
491     }
492     // Add final value if necessary.
493     if final.IsValid() {
494         t := typ.In(typ.NumIn() - 1)
495         if typ.IsVariadic() {
496             t = t.Elem()
497         }
498         argv[i] = s.validateType(final, t)
499     }
500     result := fun.Call(argv)
501     // If we have an error that is not nil, stop executi
502     if len(result) == 2 && !result[1].IsNil() {
503         s.errorf("error calling %s: %s", name, resul
504     }
505     return result[0]
506 }
507
508 // validateType guarantees that the value is valid and assign
509 func (s *state) validateType(value reflect.Value, typ reflect
510     if !value.IsValid() {
511         switch typ.Kind() {
512             case reflect.Interface, reflect.Ptr, reflect
513                 // An untyped nil interface{}. Accept
514                 // TODO: Can we delete the other typ
515                 value = reflect.Zero(typ)
516             default:
517                 s.errorf("invalid value; expected %s
518         }
519     }
520     if !value.Type().AssignableTo(typ) {
521         // Does one dereference or indirection work?
522         // do with method receivers, but that gets m
523         // are much more constrained, so it makes mo
524         // Besides, one is almost always all you need
525         switch {
526             case value.Kind() == reflect.Ptr && value.Ty
527                 value = value.Elem()
528             case reflect.PtrTo(value.Type()).AssignableT
529                 value = value.Addr()
530             default:
531                 s.errorf("wrong type for value; expe
532         }
533     }
534     return value
535 }

```

```

536
537 func (s *state) evalArg(dot reflect.Value, typ reflect.Type,
538     switch arg := n.(type) {
539     case *parse.DotNode:
540         return s.validateType(dot, typ)
541     case *parse.FieldNode:
542         return s.validateType(s.evalFieldNode(dot, a
543     case *parse.VariableNode:
544         return s.validateType(s.evalVariableNode(dot
545     }
546     switch typ.Kind() {
547     case reflect.Bool:
548         return s.evalBool(typ, n)
549     case reflect.Complex64, reflect.Complex128:
550         return s.evalComplex(typ, n)
551     case reflect.Float32, reflect.Float64:
552         return s.evalFloat(typ, n)
553     case reflect.Int, reflect.Int8, reflect.Int16, refle
554         return s.evalInteger(typ, n)
555     case reflect.Interface:
556         if typ.NumMethod() == 0 {
557             return s.evalEmptyInterface(dot, n)
558         }
559     case reflect.String:
560         return s.evalString(typ, n)
561     case reflect.Uint, reflect.Uint8, reflect.Uint16, re
562         return s.evalUnsignedInteger(typ, n)
563     }
564     s.errorf("can't handle %s for arg of type %s", n, ty
565     panic("not reached")
566 }
567
568 func (s *state) evalBool(typ reflect.Type, n parse.Node) ref
569     if n, ok := n.(*parse.BoolNode); ok {
570         value := reflect.New(typ).Elem()
571         value.SetBool(n.True)
572         return value
573     }
574     s.errorf("expected bool; found %s", n)
575     panic("not reached")
576 }
577
578 func (s *state) evalString(typ reflect.Type, n parse.Node) r
579     if n, ok := n.(*parse.StringNode); ok {
580         value := reflect.New(typ).Elem()
581         value.SetString(n.Text)
582         return value
583     }
584     s.errorf("expected string; found %s", n)

```

```

585         panic("not reached")
586     }
587
588     func (s *state) evalInteger(typ reflect.Type, n parse.Node)
589         if n, ok := n.(*parse.NumberNode); ok && n.IsInt {
590             value := reflect.New(typ).Elem()
591             value.SetInt(n.Int64)
592             return value
593         }
594         s.errorf("expected integer; found %s", n)
595         panic("not reached")
596     }
597
598     func (s *state) evalUnsignedInteger(typ reflect.Type, n pars
599         if n, ok := n.(*parse.NumberNode); ok && n.IsUint {
600             value := reflect.New(typ).Elem()
601             value.SetUint(n.Uint64)
602             return value
603         }
604         s.errorf("expected unsigned integer; found %s", n)
605         panic("not reached")
606     }
607
608     func (s *state) evalFloat(typ reflect.Type, n parse.Node) re
609         if n, ok := n.(*parse.NumberNode); ok && n.IsFloat {
610             value := reflect.New(typ).Elem()
611             value.SetFloat(n.Float64)
612             return value
613         }
614         s.errorf("expected float; found %s", n)
615         panic("not reached")
616     }
617
618     func (s *state) evalComplex(typ reflect.Type, n parse.Node)
619         if n, ok := n.(*parse.NumberNode); ok && n.IsComplex
620             value := reflect.New(typ).Elem()
621             value.SetComplex(n.Complex128)
622             return value
623         }
624         s.errorf("expected complex; found %s", n)
625         panic("not reached")
626     }
627
628     func (s *state) evalEmptyInterface(dot reflect.Value, n pars
629         switch n := n.(type) {
630         case *parse.BoolNode:
631             return reflect.ValueOf(n.True)
632         case *parse.DotNode:
633             return dot

```

```

634     case *parse.FieldNode:
635         return s.evalFieldNode(dot, n, nil, zero)
636     case *parse.IdentifierNode:
637         return s.evalFunction(dot, n.Ident, nil, zer
638     case *parse.NumberNode:
639         return s.idealConstant(n)
640     case *parse.StringNode:
641         return reflect.ValueOf(n.Text)
642     case *parse.VariableNode:
643         return s.evalVariableNode(dot, n, nil, zero)
644     }
645     s.errorf("can't handle assignment of %s to empty int
646     panic("not reached")
647 }
648
649 // indirect returns the item at the end of indirection, and
650 // We indirect through pointers and empty interfaces (only)
651 // non-empty interfaces have methods we might need.
652 func indirect(v reflect.Value) (rv reflect.Value, isNil bool
653     for ; v.Kind() == reflect.Ptr || v.Kind() == reflect
654         if v.IsNil() {
655             return v, true
656         }
657         if v.Kind() == reflect.Interface && v.NumMet
658             break
659     }
660 }
661     return v, false
662 }
663
664 // printValue writes the textual representation of the value
665 // the template.
666 func (s *state) printValue(n parse.Node, v reflect.Value) {
667     if v.Kind() == reflect.Ptr {
668         v, _ = indirect(v) // fmt.Fprint handles nil
669     }
670     if !v.IsValid() {
671         fmt.Fprint(s.wr, "<no value>")
672         return
673     }
674
675     if !v.Type().Implements(errorType) && !v.Type().Impl
676         if v.CanAddr() && (reflect.PtrTo(v.Type()).I
677             v = v.Addr()
678         } else {
679             switch v.Kind() {
680             case reflect.Chan, reflect.Func:
681                 s.errorf("can't print %s of
682             }
683         }

```

```

684         }
685         fmt.Fprint(s.wr, v.Interface())
686     }
687
688     // Types to help sort the keys in a map for reproducible out
689
690     type rvs []reflect.Value
691
692     func (x rvs) Len() int      { return len(x) }
693     func (x rvs) Swap(i, j int) { x[i], x[j] = x[j], x[i] }
694
695     type rvInts struct{ rvs }
696
697     func (x rvInts) Less(i, j int) bool { return x.rvs[i].Int()
698
699     type rvUints struct{ rvs }
700
701     func (x rvUints) Less(i, j int) bool { return x.rvs[i].Uint(
702
703     type rvFloats struct{ rvs }
704
705     func (x rvFloats) Less(i, j int) bool { return x.rvs[i].Floa
706
707     type rvStrings struct{ rvs }
708
709     func (x rvStrings) Less(i, j int) bool { return x.rvs[i].Str
710
711     // sortKeys sorts (if it can) the slice of reflect.Values, w
712     func sortKeys(v []reflect.Value) []reflect.Value {
713         if len(v) <= 1 {
714             return v
715         }
716         switch v[0].Kind() {
717             case reflect.Float32, reflect.Float64:
718                 sort.Sort(rvFloats{v})
719             case reflect.Int, reflect.Int8, reflect.Int16, refle
720                 sort.Sort(rvInts{v})
721             case reflect.String:
722                 sort.Sort(rvStrings{v})
723             case reflect.Uint, reflect.Uint8, reflect.Uint16, re
724                 sort.Sort(rvUints{v})
725         }
726         return v
727     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/template/funcs.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "bytes"
9     "fmt"
10    "io"
11    "net/url"
12    "reflect"
13    "strings"
14    "unicode"
15    "unicode/utf8"
16 )
17
18 // FuncMap is the type of the map defining the mapping from
19 // Each function must have either a single return value, or
20 // which the second has type error. In that case, if the sec
21 // argument evaluates to non-nil during execution, execution
22 // Execute returns that error.
23 type FuncMap map[string]interface{}
24
25 var builtins = FuncMap{
26     "and":      and,
27     "call":    call,
28     "html":    HTMLEscaper,
29     "index":   index,
30     "js":      JSEscaper,
31     "len":     length,
32     "not":     not,
33     "or":      or,
34     "print":   fmt.Sprint,
35     "printf":  fmt.Sprintf,
36     "println": fmt.Sprintln,
37     "urlquery": URLQueryEscaper,
38 }
39
40 var builtinFuncs = createValueFuncs(builtins)
41
```

```

42 // createValueFuncs turns a FuncMap into a map[string]reflect
43 func createValueFuncs(funcMap FuncMap) map[string]reflect.Value
44     m := make(map[string]reflect.Value)
45     addValueFuncs(m, funcMap)
46     return m
47 }
48
49 // addValueFuncs adds to values the functions in funcs, conv
50 func addValueFuncs(out map[string]reflect.Value, in FuncMap)
51     for name, fn := range in {
52         v := reflect.ValueOf(fn)
53         if v.Kind() != reflect.Func {
54             panic("value for " + name + " not a
55         }
56         if !goodFunc(v.Type()) {
57             panic(fmt.Errorf("can't handle multi
58         }
59         out[name] = v
60     }
61 }
62
63 // addFuncs adds to values the functions in funcs. It does n
64 // call addValueFuncs first.
65 func addFuncs(out, in FuncMap) {
66     for name, fn := range in {
67         out[name] = fn
68     }
69 }
70
71 // goodFunc checks that the function or method has the right
72 func goodFunc(typ reflect.Type) bool {
73     // We allow functions with 1 result or 2 results whe
74     switch {
75     case typ.NumOut() == 1:
76         return true
77     case typ.NumOut() == 2 && typ.Out(1) == errorType:
78         return true
79     }
80     return false
81 }
82
83 // findFunction looks for a function in the template, and gl
84 func findFunction(name string, tmpl *Template) (reflect.Valu
85     if tmpl != nil && tmpl.common != nil {
86         if fn := tmpl.execFuncs[name]; fn.IsValid()
87             return fn, true
88     }
89 }
90     if fn := builtinFuncs[name]; fn.IsValid() {
91         return fn, true

```

```

92         }
93         return reflect.Value{}, false
94     }
95
96     // Indexing.
97
98     // index returns the result of indexing its first argument b
99     // arguments. Thus "index x 1 2 3" is, in Go syntax, x[1][2
100    // indexed item must be a map, slice, or array.
101    func index(item interface{}, indices ...interface{}) (interf
102        v := reflect.ValueOf(item)
103        for _, i := range indices {
104            index := reflect.ValueOf(i)
105            var isNil bool
106            if v, isNil = indirect(v); isNil {
107                return nil, fmt.Errorf("index of nil
108            }
109            switch v.Kind() {
110            case reflect.Array, reflect.Slice:
111                var x int64
112                switch index.Kind() {
113                case reflect.Int, reflect.Int8, refl
114                    x = index.Int()
115                case reflect.Uint, reflect.Uint8, re
116                    x = int64(index.Uint())
117                default:
118                    return nil, fmt.Errorf("cann
119                }
120                if x < 0 || x >= int64(v.Len()) {
121                    return nil, fmt.Errorf("inde
122                }
123                v = v.Index(int(x))
124            case reflect.Map:
125                if !index.Type().AssignableTo(v.Type
126                    return nil, fmt.Errorf("%s i
127                }
128                if x := v.MapIndex(index); x.IsValid
129                    v = x
130                } else {
131                    v = reflect.Zero(v.Type()).Ke
132                }
133            default:
134                return nil, fmt.Errorf("can't index
135            }
136        }
137        return v.Interface(), nil
138    }
139
140    // Length

```

```

141
142 // length returns the length of the item, with an error if i
143 func length(item interface{}) (int, error) {
144     v, isNil := indirect(reflect.ValueOf(item))
145     if isNil {
146         return 0, fmt.Errorf("len of nil pointer")
147     }
148     switch v.Kind() {
149     case reflect.Array, reflect.Chan, reflect.Map, refle
150         return v.Len(), nil
151     }
152     return 0, fmt.Errorf("len of type %s", v.Type())
153 }
154
155 // Function invocation
156
157 // call returns the result of evaluating the the first argum
158 // The function must return 1 result, or 2 results, the seco
159 func call(fn interface{}, args ...interface{}) (interface{},
160     v := reflect.ValueOf(fn)
161     typ := v.Type()
162     if typ.Kind() != reflect.Func {
163         return nil, fmt.Errorf("non-function of type
164     }
165     if !goodFunc(typ) {
166         return nil, fmt.Errorf("function called with
167     }
168     numIn := typ.NumIn()
169     var dddType reflect.Type
170     if typ.IsVariadic() {
171         if len(args) < numIn-1 {
172             return nil, fmt.Errorf("wrong number
173         }
174         dddType = typ.In(numIn - 1).Elem()
175     } else {
176         if len(args) != numIn {
177             return nil, fmt.Errorf("wrong number
178         }
179     }
180     argv := make([]reflect.Value, len(args))
181     for i, arg := range args {
182         value := reflect.ValueOf(arg)
183         // Compute the expected type. Clumsy because
184         var argType reflect.Type
185         if !typ.IsVariadic() || i < numIn-1 {
186             argType = typ.In(i)
187         } else {
188             argType = dddType
189         }

```

```

190         if !value.Type().AssignableTo(argType) {
191             return nil, fmt.Errorf("arg %d has t
192         }
193         argv[i] = reflect.ValueOf(arg)
194     }
195     result := v.Call(argv)
196     if len(result) == 2 {
197         return result[0].Interface(), result[1].Inte
198     }
199     return result[0].Interface(), nil
200 }
201
202 // Boolean logic.
203
204 func truth(a interface{}) bool {
205     t, _ := isTrue(reflect.ValueOf(a))
206     return t
207 }
208
209 // and computes the Boolean AND of its arguments, returning
210 // the first false argument it encounters, or the last argum
211 func and(arg0 interface{}, args ...interface{}) interface{}
212     if !truth(arg0) {
213         return arg0
214     }
215     for i := range args {
216         arg0 = args[i]
217         if !truth(arg0) {
218             break
219         }
220     }
221     return arg0
222 }
223
224 // or computes the Boolean OR of its arguments, returning
225 // the first true argument it encounters, or the last argume
226 func or(arg0 interface{}, args ...interface{}) interface{} {
227     if truth(arg0) {
228         return arg0
229     }
230     for i := range args {
231         arg0 = args[i]
232         if truth(arg0) {
233             break
234         }
235     }
236     return arg0
237 }
238
239 // not returns the Boolean negation of its argument.

```

```

240 func not(arg interface{}) (truth bool) {
241     truth, _ = isTrue(reflect.ValueOf(arg))
242     return !truth
243 }
244
245 // HTML escaping.
246
247 var (
248     htmlQuot = []byte("&#34;") // shorter than "&quot;"
249     htmlApos = []byte("&#39;") // shorter than "&apos;"
250     htmlAmp  = []byte("&amp;")
251     htmlLt   = []byte("&lt;")
252     htmlGt   = []byte("&gt;")
253 )
254
255 // HTML Escape writes to w the escaped HTML equivalent of the
256 func HTML Escape(w io.Writer, b []byte) {
257     last := 0
258     for i, c := range b {
259         var html []byte
260         switch c {
261             case '"':
262                 html = htmlQuot
263             case '\':
264                 html = htmlApos
265             case '&':
266                 html = htmlAmp
267             case '<':
268                 html = htmlLt
269             case '>':
270                 html = htmlGt
271             default:
272                 continue
273         }
274         w.Write(b[last:i])
275         w.Write(html)
276         last = i + 1
277     }
278     w.Write(b[last:])
279 }
280
281 // HTML EscapeString returns the escaped HTML equivalent of t
282 func HTML EscapeString(s string) string {
283     // Avoid allocation if we can.
284     if strings.IndexAny(s, `'"&<>`) < 0 {
285         return s
286     }
287     var b bytes.Buffer
288     HTML Escape(&b, []byte(s))

```

```

289         return b.String()
290     }
291
292     // HTMLEscaper returns the escaped HTML equivalent of the te
293     // representation of its arguments.
294     func HTMLEscaper(args ...interface{}) string {
295         ok := false
296         var s string
297         if len(args) == 1 {
298             s, ok = args[0].(string)
299         }
300         if !ok {
301             s = fmt.Sprint(args...)
302         }
303         return HTMLEscapeString(s)
304     }
305
306     // JavaScript escaping.
307
308     var (
309         jsLowUni = []byte(`\u00`)
310         hex      = []byte("0123456789ABCDEF")
311
312         jsBackslash = []byte(`\\`)
313         jsApos      = []byte(`\'`)
314         jsQuot     = []byte(`\"`)
315         jsLt       = []byte(`\x3C`)
316         jsGt       = []byte(`\x3E`)
317     )
318
319     // JSEscape writes to w the escaped JavaScript equivalent of
320     func JSEscape(w io.Writer, b []byte) {
321         last := 0
322         for i := 0; i < len(b); i++ {
323             c := b[i]
324
325             if !jsIsSpecial(rune(c)) {
326                 // fast path: nothing to do
327                 continue
328             }
329             w.Write(b[last:i])
330
331             if c < utf8.RuneSelf {
332                 // Quotes, slashes and angle bracket
333                 // Control characters get written as
334                 switch c {
335                 case `\\`:
336                     w.Write(jsBackslash)
337                 case `\'`:

```

```

338         w.Write(jsApos)
339     case '"':
340         w.Write(jsQuot)
341     case '<':
342         w.Write(jsLt)
343     case '>':
344         w.Write(jsGt)
345     default:
346         w.Write(jsLowUni)
347         t, b := c>>4, c&0xf
348         w.Write(hex[t : t+1])
349         w.Write(hex[b : b+1])
350     }
351 } else {
352     // Unicode rune.
353     r, size := utf8.DecodeRune(b[i:])
354     if unicode.IsPrint(r) {
355         w.Write(b[i : i+size])
356     } else {
357         fmt.Fprintf(w, "\\u%04X", r)
358     }
359     i += size - 1
360 }
361 last = i + 1
362 }
363 w.Write(b[last:])
364 }
365
366 // JSEscapeString returns the escaped JavaScript equivalent
367 func JSEscapeString(s string) string {
368     // Avoid allocation if we can.
369     if strings.IndexFunc(s, jsIsSpecial) < 0 {
370         return s
371     }
372     var b bytes.Buffer
373     JSEscape(&b, []byte(s))
374     return b.String()
375 }
376
377 func jsIsSpecial(r rune) bool {
378     switch r {
379     case '\\', '\'', '"', '<', '>':
380         return true
381     }
382     return r < ' ' || utf8.RuneSelf <= r
383 }
384
385 // JSEscaper returns the escaped JavaScript equivalent of th
386 // representation of its arguments.
387 func JSEscaper(args ...interface{}) string {

```

```

388         ok := false
389         var s string
390         if len(args) == 1 {
391             s, ok = args[0].(string)
392         }
393         if !ok {
394             s = fmt.Sprint(args...)
395         }
396         return JSEscapeString(s)
397     }
398
399     // URLQueryEscaper returns the escaped value of the textual
400     // its arguments in a form suitable for embedding in a URL q
401     func URLQueryEscaper(args ...interface{}) string {
402         s, ok := "", false
403         if len(args) == 1 {
404             s, ok = args[0].(string)
405         }
406         if !ok {
407             s = fmt.Sprint(args...)
408         }
409         return url.QueryEscape(s)
410     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/template/helper.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Helper functions to make constructing templates easier.
6
7 package template
8
9 import (
10     "fmt"
11     "io/ioutil"
12     "path/filepath"
13 )
14
15 // Functions and methods to parse templates.
16
17 // Must is a helper that wraps a call to a function returnin
18 // and panics if the error is non-nil. It is intended for us
19 // initializations such as
20 //     var t = template.Must(template.New("name").Parse("te
21 func Must(t *Template, err error) *Template {
22     if err != nil {
23         panic(err)
24     }
25     return t
26 }
27
28 // ParseFiles creates a new Template and parses the template
29 // the named files. The returned template's name will have t
30 // (parsed) contents of the first file. There must be at lea
31 // If an error occurs, parsing stops and the returned *Templ
32 func ParseFiles(filenamees ...string) (*Template, error) {
33     return parseFiles(nil, filenamees...)
34 }
35
36 // ParseFiles parses the named files and associates the resu
37 // t. If an error occurs, parsing stops and the returned tem
38 // otherwise it is t. There must be at least one file.
39 func (t *Template) ParseFiles(filenamees ...string) (*Templat
40     return parseFiles(t, filenamees...)
41 }
```

```

42
43 // parseFiles is the helper for the method and function. If
44 // template is nil, it is created from the first file.
45 func parseFiles(t *Template, filenames ...string) (*Template
46     if len(filenames) == 0 {
47         // Not really a problem, but be consistent.
48         return nil, fmt.Errorf("template: no files n
49     }
50     for _, filename := range filenames {
51         b, err := ioutil.ReadFile(filename)
52         if err != nil {
53             return nil, err
54         }
55         s := string(b)
56         name := filepath.Base(filename)
57         // First template becomes return value if no
58         // and we use that one for subsequent New ca
59         // all the templates together. Also, if this
60         // as t, this file becomes the contents of t
61         // t, err := New(name).Funcs(XXX).ParseFile
62         // works. Otherwise we create a new template
63         var tmpl *Template
64         if t == nil {
65             t = New(name)
66         }
67         if name == t.Name() {
68             tmpl = t
69         } else {
70             tmpl = t.New(name)
71         }
72         _, err = tmpl.Parse(s)
73         if err != nil {
74             return nil, err
75         }
76     }
77     return t, nil
78 }
79
80 // ParseGlob creates a new Template and parses the template
81 // files identified by the pattern, which must match at leas
82 // returned template will have the (base) name and (parsed)
83 // first file matched by the pattern. ParseGlob is equivalen
84 // ParseFiles with the list of files matched by the pattern.
85 func ParseGlob(pattern string) (*Template, error) {
86     return parseGlob(nil, pattern)
87 }
88
89 // ParseGlob parses the template definitions in the files id
90 // pattern and associates the resulting templates with t. Th
91 // processed by filepath.Glob and must match at least one fi

```

```
92 // equivalent to calling t.ParseFiles with the list of files
93 // pattern.
94 func (t *Template) ParseGlob(pattern string) (*Template, err
95     return parseGlob(t, pattern)
96 }
97
98 // parseGlob is the implementation of the function and metho
99 func parseGlob(t *Template, pattern string) (*Template, erro
100     filenames, err := filepath.Glob(pattern)
101     if err != nil {
102         return nil, err
103     }
104     if len(filenames) == 0 {
105         return nil, fmt.Errorf("template: pattern ma
106     }
107     return parseFiles(t, filenames...)
108 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/template/template.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package template
6
7 import (
8     "fmt"
9     "reflect"
10    "text/template/parse"
11 )
12
13 // common holds the information shared by related templates.
14 type common struct {
15     tmpl map[string]*Template
16     // We use two maps, one for parsing and one for exec
17     // This separation makes the API cleaner since it do
18     // expose reflection to the client.
19     parseFuncs FuncMap
20     execFuncs  map[string]reflect.Value
21 }
22
23 // Template is the representation of a parsed template. The
24 // field is exported only for use by html/template and shoul
25 // as unexported by all other clients.
26 type Template struct {
27     name string
28     *parse.Tree
29     *common
30     leftDelim string
31     rightDelim string
32 }
33
34 // New allocates a new template with the given name.
35 func New(name string) *Template {
36     return &Template{
37         name: name,
38     }
39 }
40
41 // Name returns the name of the template.
```

```

42 func (t *Template) Name() string {
43     return t.name
44 }
45
46 // New allocates a new template associated with the given on
47 // delimiters. The association, which is transitive, allows
48 // invoke another with a {{template}} action.
49 func (t *Template) New(name string) *Template {
50     t.init()
51     return &Template{
52         name:      name,
53         common:    t.common,
54         leftDelim: t.leftDelim,
55         rightDelim: t.rightDelim,
56     }
57 }
58
59 func (t *Template) init() {
60     if t.common == nil {
61         t.common = new(common)
62         t.tmpl = make(map[string]*Template)
63         t.parseFuncs = make(FuncMap)
64         t.execFuncs = make(map[string]reflect.Value)
65     }
66 }
67
68 // Clone returns a duplicate of the template, including all
69 // templates. The actual representation is not copied, but t
70 // associated templates is, so further calls to Parse in the
71 // templates to the copy but not to the original. Clone can
72 // common templates and use them with variant definitions fo
73 // by adding the variants after the clone is made.
74 func (t *Template) Clone() (*Template, error) {
75     nt := t.copy(nil)
76     nt.init()
77     nt.tmpl[t.name] = nt
78     for k, v := range t.tmpl {
79         if k == t.name { // Already installed.
80             continue
81         }
82         // The associated templates share nt's commo
83         tmpl := v.copy(nt.common)
84         nt.tmpl[k] = tmpl
85     }
86     for k, v := range t.parseFuncs {
87         nt.parseFuncs[k] = v
88     }
89     for k, v := range t.execFuncs {
90         nt.execFuncs[k] = v
91     }

```

```

92         return nt, nil
93     }
94
95     // copy returns a shallow copy of t, with common set to the
96     func (t *Template) copy(c *common) *Template {
97         nt := New(t.name)
98         nt.Tree = t.Tree
99         nt.common = c
100        nt.leftDelim = t.leftDelim
101        nt.rightDelim = t.rightDelim
102        return nt
103    }
104
105    // AddParseTree creates a new template with the name and par
106    // and associates it with t.
107    func (t *Template) AddParseTree(name string, tree *parse.Tree
108        if t.tmpl[name] != nil {
109            return nil, fmt.Errorf("template: redefiniti
110        }
111        nt := t.New(name)
112        nt.Tree = tree
113        t.tmpl[name] = nt
114        return nt, nil
115    }
116
117    // Templates returns a slice of the templates associated wit
118    // itself.
119    func (t *Template) Templates() []*Template {
120        // Return a slice so we don't expose the map.
121        m := make([]*Template, 0, len(t.tmpl))
122        for _, v := range t.tmpl {
123            m = append(m, v)
124        }
125        return m
126    }
127
128    // Delims sets the action delimiters to the specified string
129    // subsequent calls to Parse, ParseFiles, or ParseGlob. Nest
130    // definitions will inherit the settings. An empty delimiter
131    // corresponding default: {{ or }}.
132    // The return value is the template, so calls can be chained
133    func (t *Template) Delims(left, right string) *Template {
134        t.leftDelim = left
135        t.rightDelim = right
136        return t
137    }
138
139    // Funcs adds the elements of the argument map to the templa
140    // It panics if a value in the map is not a function with ap

```

```

141 // type. However, it is legal to overwrite elements of the m
142 // value is the template, so calls can be chained.
143 func (t *Template) Funcs(funcMap FuncMap) *Template {
144     t.init()
145     addValueFuncs(t.execFuncs, funcMap)
146     addFuncs(t.parseFuncs, funcMap)
147     return t
148 }
149
150 // Lookup returns the template with the given name that is a
151 // or nil if there is no such template.
152 func (t *Template) Lookup(name string) *Template {
153     if t.common == nil {
154         return nil
155     }
156     return t.tmpl[name]
157 }
158
159 // Parse parses a string into a template. Nested template de
160 // associated with the top-level template t. Parse may be ca
161 // to parse definitions of templates to associate with t. It
162 // resulting template is non-empty (contains content other t
163 // definitions) and would replace a non-empty template with
164 // (In multiple calls to Parse with the same receiver templa
165 // can contain text other than space, comments, and template
166 func (t *Template) Parse(text string) (*Template, error) {
167     t.init()
168     trees, err := parse.Parse(t.name, text, t.leftDelim,
169     if err != nil {
170         return nil, err
171     }
172     // Add the newly parsed trees, including the one for
173     for name, tree := range trees {
174         // If the name we parsed is the name of this
175         // The associate method checks it's not a re
176         tmpl := t
177         if name != t.name {
178             tmpl = t.New(name)
179         }
180         // Even if t == tmpl, we need to install it
181         if replace, err := t.associate(tmpl, tree);
182             return nil, err
183         } else if replace {
184             tmpl.Tree = tree
185         }
186         tmpl.leftDelim = t.leftDelim
187         tmpl.rightDelim = t.rightDelim
188     }
189     return t, nil

```

```

190 }
191
192 // associate installs the new template into the group of templates
193 // with t. It is an error to reuse a name except to overwrite
194 // template. The two are already known to share the common symbols
195 // The boolean return value reports whether to store this tree
196 func (t *Template) associate(new *Template, tree *parse.Tree) bool {
197     if new.common != t.common {
198         panic("internal error: associate not common")
199     }
200     name := new.name
201     if old := t.tmpl[name]; old != nil {
202         oldIsEmpty := parse.IsEmptyTree(old.Root)
203         newIsEmpty := parse.IsEmptyTree(tree.Root)
204         if newIsEmpty {
205             // Whether old is empty or not, new
206             // return false, nil
207         }
208         if !oldIsEmpty {
209             return false, fmt.Errorf("template:
210         }
211     }
212     t.tmpl[name] = new
213     return true, nil
214 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/template/parse/lex.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package parse
6
7 import (
8     "fmt"
9     "strings"
10    "unicode"
11    "unicode/utf8"
12 )
13
14 // item represents a token or text string returned from the
15 type item struct {
16     typ itemType
17     val string
18 }
19
20 func (i item) String() string {
21     switch {
22     case i.typ == itemEOF:
23         return "EOF"
24     case i.typ == itemError:
25         return i.val
26     case i.typ > itemKeyword:
27         return fmt.Sprintf("<%s>", i.val)
28     case len(i.val) > 10:
29         return fmt.Sprintf("%.10q...", i.val)
30     }
31     return fmt.Sprintf("%q", i.val)
32 }
33
34 // itemType identifies the type of lex items.
35 type itemType int
36
37 const (
38     itemError      itemType = iota // error occurred;
39     itemBool       // boolean constant
40     itemChar       // printable ASCII
41     itemCharConstant // character constant
```

```

42     itemComplex           // complex constant
43     itemColonEquals      // colon-equals (':
44     itemEOF
45     itemField            // alphanumeric identifier, starting
46     itemIdentifier       // alphanumeric identifier
47     itemLeftDelim        // left action delimiter
48     itemNumber           // simple number, including imaginary
49     itemPipe             // pipe symbol
50     itemRawString        // raw quoted string (includes quotes
51     itemRightDelim       // right action delimiter
52     itemString           // quoted string (includes quotes)
53     itemText             // plain text
54     itemVariable         // variable starting with '$', such a
55     // Keywords appear after all the rest.
56     itemKeyword          // used only to delimit the keywords
57     itemDot              // the cursor, spelled '.'.
58     itemDefine           // define keyword
59     itemElse             // else keyword
60     itemEnd              // end keyword
61     itemIf               // if keyword
62     itemRange            // range keyword
63     itemTemplate         // template keyword
64     itemWith             // with keyword
65 )
66
67 // Make the types prettyprint.
68 var itemName = map[itemType]string{
69     itemError:           "error",
70     itemBool:            "bool",
71     itemChar:            "char",
72     itemCharConstant:   "charconst",
73     itemComplex:         "complex",
74     itemColonEquals:     ":",
75     itemEOF:             "EOF",
76     itemField:           "field",
77     itemIdentifier:      "identifier",
78     itemLeftDelim:       "left delim",
79     itemNumber:          "number",
80     itemPipe:            "pipe",
81     itemRawString:       "raw string",
82     itemRightDelim:      "right delim",
83     itemString:          "string",
84     itemVariable:        "variable",
85     // keywords
86     itemDot:             ".",
87     itemDefine:          "define",
88     itemElse:            "else",
89     itemIf:              "if",
90     itemEnd:             "end",
91     itemRange:           "range",

```

```

92         itemTemplate: "template",
93         itemWith:      "with",
94     }
95
96     func (i itemType) String() string {
97         s := itemName[i]
98         if s == "" {
99             return fmt.Sprintf("item%d", int(i))
100        }
101        return s
102    }
103
104    var key = map[string]itemType{
105        ".":      itemDot,
106        "define": itemDefine,
107        "else":   itemElse,
108        "end":    itemEnd,
109        "if":     itemIf,
110        "range": itemRange,
111        "template": itemTemplate,
112        "with":   itemWith,
113    }
114
115    const eof = -1
116
117    // stateFn represents the state of the scanner as a function
118    type stateFn func(*lexer) stateFn
119
120    // lexer holds the state of the scanner.
121    type lexer struct {
122        name      string // the name of the input; used
123        input    string // the string being scanned.
124        leftDelim string // start of action.
125        rightDelim string // end of action.
126        state    stateFn // the next lexing function to
127        pos      int    // current position in the input
128        start    int    // start position of this item.
129        width    int    // width of last rune read from
130        items    chan item // channel of scanned items.
131    }
132
133    // next returns the next rune in the input.
134    func (l *lexer) next() (r rune) {
135        if l.pos >= len(l.input) {
136            l.width = 0
137            return eof
138        }
139        r, l.width = utf8.DecodeRuneInString(l.input[l.pos:])
140        l.pos += l.width

```

```

141         return r
142     }
143
144     // peek returns but does not consume the next rune in the in
145     func (l *lexer) peek() rune {
146         r := l.next()
147         l.backup()
148         return r
149     }
150
151     // backup steps back one rune. Can only be called once per c
152     func (l *lexer) backup() {
153         l.pos -= l.width
154     }
155
156     // emit passes an item back to the client.
157     func (l *lexer) emit(t itemType) {
158         l.items <- item{t, l.input[l.start:l.pos]}
159         l.start = l.pos
160     }
161
162     // ignore skips over the pending input before this point.
163     func (l *lexer) ignore() {
164         l.start = l.pos
165     }
166
167     // accept consumes the next rune if it's from the valid set.
168     func (l *lexer) accept(valid string) bool {
169         if strings.IndexRune(valid, l.next()) >= 0 {
170             return true
171         }
172         l.backup()
173         return false
174     }
175
176     // acceptRun consumes a run of runes from the valid set.
177     func (l *lexer) acceptRun(valid string) {
178         for strings.IndexRune(valid, l.next()) >= 0 {
179             }
180         l.backup()
181     }
182
183     // lineNumber reports which line we're on. Doing it this way
184     // means we don't have to worry about peek double counting.
185     func (l *lexer) lineNumber() int {
186         return 1 + strings.Count(l.input[:l.pos], "\n")
187     }
188
189     // error returns an error token and terminates the scan by p

```

```

190 // back a nil pointer that will be the next state, terminati
191 func (l *lexer) errorf(format string, args ...interface{}) s
192     l.items <- item{itemError, fmt.Sprintf(format, args.
193     return nil
194 }
195
196 // nextItem returns the next item from the input.
197 func (l *lexer) nextItem() item {
198     for {
199         select {
200             case item := <-l.items:
201                 return item
202             default:
203                 l.state = l.state(1)
204             }
205         }
206     panic("not reached")
207 }
208
209 // lex creates a new scanner for the input string.
210 func lex(name, input, left, right string) *lexer {
211     if left == "" {
212         left = leftDelim
213     }
214     if right == "" {
215         right = rightDelim
216     }
217     l := &lexer{
218         name:      name,
219         input:      input,
220         leftDelim: left,
221         rightDelim: right,
222         state:     lexText,
223         items:     make(chan item, 2), // Two items
224     }
225     return l
226 }
227
228 // state functions
229
230 const (
231     leftDelim    = "{{"
232     rightDelim   = "}}"
233     leftComment  = "/*"
234     rightComment = "*/"
235 )
236
237 // lexText scans until an opening action delimiter, "{{"
238 func lexText(l *lexer) stateFn {
239     for {

```

```

240         if strings.HasPrefix(l.input[l.pos:], l.left
241             if l.pos > l.start {
242                 l.emit(itemText)
243             }
244             return lexLeftDelim
245         }
246         if l.next() == eof {
247             break
248         }
249     }
250     // Correctly reached EOF.
251     if l.pos > l.start {
252         l.emit(itemText)
253     }
254     l.emit(itemEOF)
255     return nil
256 }
257
258 // lexLeftDelim scans the left delimiter, which is known to
259 func lexLeftDelim(l *lexer) stateFn {
260     if strings.HasPrefix(l.input[l.pos:], l.leftDelim+l.r
261         return lexComment
262     }
263     l.pos += len(l.leftDelim)
264     l.emit(itemLeftDelim)
265     return lexInsideAction
266 }
267
268 // lexComment scans a comment. The left comment marker is kn
269 func lexComment(l *lexer) stateFn {
270     i := strings.Index(l.input[l.pos:], rightComment+l.r
271     if i < 0 {
272         return l.errorf("unclosed comment")
273     }
274     l.pos += i + len(rightComment) + len(l.rightDelim)
275     l.ignore()
276     return lexText
277 }
278
279 // lexRightDelim scans the right delimiter, which is known t
280 func lexRightDelim(l *lexer) stateFn {
281     l.pos += len(l.rightDelim)
282     l.emit(itemRightDelim)
283     return lexText
284 }
285
286 // lexInsideAction scans the elements inside action delimit
287 func lexInsideAction(l *lexer) stateFn {
288     // Either number, quoted string, or identifier.

```

```

289 // Spaces separate and are ignored.
290 // Pipe symbols separate and are emitted.
291 if strings.HasPrefix(l.input[l.pos:], l.rightDelim)
292     return lexRightDelim
293 }
294 switch r := l.next(); {
295 case r == eof || r == '\n':
296     return l.errorf("unclosed action")
297 case isSpace(r):
298     l.ignore()
299 case r == ':':
300     if l.next() != '=' {
301         return l.errorf("expected :=")
302     }
303     l.emit(itemColonEquals)
304 case r == '|':
305     l.emit(itemPipe)
306 case r == '"':
307     return lexQuote
308 case r == '`':
309     return lexRawQuote
310 case r == '$':
311     return lexIdentifier
312 case r == '\\':
313     return lexChar
314 case r == '.':
315     // special look-ahead for ".field" so we don
316     if l.pos < len(l.input) {
317         r := l.input[l.pos]
318         if r < '0' || '9' < r {
319             return lexIdentifier // item
320         }
321     }
322     fallthrough // '.' can start a number.
323 case r == '+' || r == '-' || ('0' <= r && r <= '9'):
324     l.backup()
325     return lexNumber
326 case isAlphaNumeric(r):
327     l.backup()
328     return lexIdentifier
329 case r <= unicode.MaxASCII && unicode.IsPrint(r):
330     l.emit(itemChar)
331     return lexInsideAction
332 default:
333     return l.errorf("unrecognized character in a
334 }
335 return lexInsideAction
336 }
337

```

```

338 // lexIdentifier scans an alphanumeric or field.
339 func lexIdentifier(l *lexer) stateFn {
340 Loop:
341     for {
342         switch r := l.next(); {
343         case isAlphaNumeric(r):
344             // absorb.
345         case r == '.' && (l.input[l.start] == '.' ||
346             // field chaining; absorb into one t
347         default:
348             l.backup()
349             word := l.input[l.start:l.pos]
350             if !l.atTerminator() {
351                 return l.errorf("unexpected
352             }
353             switch {
354             case key[word] > itemKeyword:
355                 l.emit(key[word])
356             case word[0] == '.':
357                 l.emit(itemField)
358             case word[0] == '$':
359                 l.emit(itemVariable)
360             case word == "true", word == "false"
361                 l.emit(itemBool)
362             default:
363                 l.emit(itemIdentifier)
364             }
365             break Loop
366         }
367     }
368     return lexInsideAction
369 }
370
371 // atTerminator reports whether the input is at valid termin
372 // appear after an identifier. Mostly to catch cases like "$
373 // acceptable without a space, in case we decide one day to
374 // arithmetic.
375 func (l *lexer) atTerminator() bool {
376     r := l.peek()
377     if isSpace(r) {
378         return true
379     }
380     switch r {
381     case eof, ',', '|', ':':
382         return true
383     }
384     // Does r start the delimiter? This can be ambiguous
385     // succeed but should fail) but only in extremely ra
386     // bad choice of delimiter.
387     if rd, _ := utf8.DecodeRuneInString(l.rightDelim); r

```

```

388         return true
389     }
390     return false
391 }
392
393 // lexChar scans a character constant. The initial quote is
394 // scanned. Syntax checking is done by the parse.
395 func lexChar(l *lexer) stateFn {
396 Loop:
397     for {
398         switch l.next() {
399             case '\\':
400                 if r := l.next(); r != eof && r != '
401                     break
402                 }
403                 fallthrough
404             case eof, '\n':
405                 return l.errorf("unterminated charac
406             case '\':
407                 break Loop
408         }
409     }
410     l.emit(itemCharConstant)
411     return lexInsideAction
412 }
413
414 // lexNumber scans a number: decimal, octal, hex, float, or
415 // isn't a perfect number scanner - for instance it accepts
416 // and "089" - but when it's wrong the input is invalid and
417 // strconv) will notice.
418 func lexNumber(l *lexer) stateFn {
419     if !l.scanNumber() {
420         return l.errorf("bad number syntax: %q", l.i
421     }
422     if sign := l.peek(); sign == '+' || sign == '-' {
423         // Complex: 1+2i. No spaces, must end in 'i
424         if !l.scanNumber() || l.input[l.pos-1] != 'i
425             return l.errorf("bad number syntax:
426         }
427         l.emit(itemComplex)
428     } else {
429         l.emit(itemNumber)
430     }
431     return lexInsideAction
432 }
433
434 func (l *lexer) scanNumber() bool {
435     // Optional leading sign.
436     l.accept("+ -")

```

```

437         // Is it hex?
438         digits := "0123456789"
439         if l.accept("0") && l.accept("xX") {
440             digits = "0123456789abcdefABCDEF"
441         }
442         l.acceptRun(digits)
443         if l.accept(".") {
444             l.acceptRun(digits)
445         }
446         if l.accept("eE") {
447             l.accept("+ -")
448             l.acceptRun("0123456789")
449         }
450         // Is it imaginary?
451         l.accept("i")
452         // Next thing mustn't be alphanumeric.
453         if isAlphaNumeric(l.peek()) {
454             l.next()
455             return false
456         }
457         return true
458     }
459
460     // lexQuote scans a quoted string.
461     func lexQuote(l *lexer) stateFn {
462     Loop:
463         for {
464             switch l.next() {
465             case '\\':
466                 if r := l.next(); r != eof && r != '
467                     break
468                 }
469                 fallthrough
470             case eof, '\n':
471                 return l.errorf("unterminated quoted
472             case '":
473                 break Loop
474             }
475         }
476         l.emit(itemString)
477         return lexInsideAction
478     }
479
480     // lexRawQuote scans a raw quoted string.
481     func lexRawQuote(l *lexer) stateFn {
482     Loop:
483         for {
484             switch l.next() {
485             case eof, '\n':

```

```

486             return l.errorf("unterminated raw qu
487             case '`':
488                 break Loop
489             }
490         }
491         l.emit(itemRawString)
492         return lexInsideAction
493     }
494
495     // isSpace reports whether r is a space character.
496     func isSpace(r rune) bool {
497         switch r {
498             case ' ', '\t', '\n', '\r':
499                 return true
500         }
501         return false
502     }
503
504     // isAlphaNumeric reports whether r is an alphabetic, digit,
505     func isAlphaNumeric(r rune) bool {
506         return r == '_' || unicode.IsLetter(r) || unicode.Is
507     }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/template/parse/node.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Parse nodes.
6
7 package parse
8
9 import (
10     "bytes"
11     "fmt"
12     "strconv"
13     "strings"
14 )
15
16 // A node is an element in the parse tree. The interface is
17 type Node interface {
18     Type() NodeType
19     String() string
20     // Copy does a deep copy of the Node and all its children.
21     // To avoid type assertions, some XxxNodes also have
22     // CopyXxx methods that return *XxxNode.
23     Copy() Node
24 }
25
26 // NodeType identifies the type of a parse tree node.
27 type NodeType int
28
29 // Type returns itself and provides an easy default implementation
30 // for embedding in a Node. Embedded in all non-trivial Node
31 func (t NodeType) Type() NodeType {
32     return t
33 }
34
35 const (
36     NodeText      NodeType = iota // Plain text.
37     NodeAction    // A simple action such as {{.}}
38     NodeBool      // A boolean constant
39     NodeCommand   // An element of a pipeline
40     NodeDot       // The cursor, dot.
41     nodeElse     // An else action. No
```

```

42         nodeEnd                // An end action. Not
43         NodeField              // A field or method
44         NodeIdentifier         // An identifier; alw
45         NodeIf                 // An if action.
46         NodeList              // A list of Nodes.
47         NodeNumber            // A numerical consta
48         NodePipe              // A pipeline of comm
49         NodeRange             // A range action.
50         NodeString            // A string constant.
51         NodeTemplate          // A template invocat
52         NodeVariable          // A $ variable.
53         NodeWith              // A with action.
54     )
55
56 // Nodes.
57
58 // ListNode holds a sequence of nodes.
59 type ListNode struct {
60     NodeType
61     Nodes []Node // The element nodes in lexical order.
62 }
63
64 func newList() *ListNode {
65     return &ListNode{NodeType: NodeList}
66 }
67
68 func (l *ListNode) append(n Node) {
69     l.Nodes = append(l.Nodes, n)
70 }
71
72 func (l *ListNode) String() string {
73     b := new(bytes.Buffer)
74     for _, n := range l.Nodes {
75         fmt.Fprint(b, n)
76     }
77     return b.String()
78 }
79
80 func (l *ListNode) CopyList() *ListNode {
81     if l == nil {
82         return l
83     }
84     n := newList()
85     for _, elem := range l.Nodes {
86         n.append(elem.Copy())
87     }
88     return n
89 }
90
91 func (l *ListNode) Copy() Node {

```

```

92         return l.CopyList()
93     }
94
95     // TextNode holds plain text.
96     type TextNode struct {
97         NodeType
98         Text []byte // The text; may span newlines.
99     }
100
101     func newText(text string) *TextNode {
102         return &TextNode{NodeType: NodeText, Text: []byte(text)}
103     }
104
105     func (t *TextNode) String() string {
106         return fmt.Sprintf("%q", t.Text)
107     }
108
109     func (t *TextNode) Copy() Node {
110         return &TextNode{NodeType: NodeText, Text: append([], t.Text)}
111     }
112
113     // PipeNode holds a pipeline with optional declaration
114     type PipeNode struct {
115         NodeType
116         Line int // The line number in the input
117         Decl []*VariableNode // Variable declarations in lex
118         Cmds []*CommandNode // The commands in lexical order
119     }
120
121     func newPipeline(line int, decl []*VariableNode) *PipeNode {
122         return &PipeNode{NodeType: NodePipe, Line: line, Decl: decl}
123     }
124
125     func (p *PipeNode) append(command *CommandNode) {
126         p.Cmds = append(p.Cmds, command)
127     }
128
129     func (p *PipeNode) String() string {
130         s := ""
131         if len(p.Decl) > 0 {
132             for i, v := range p.Decl {
133                 if i > 0 {
134                     s += ", "
135                 }
136                 s += v.String()
137             }
138             s += " := "
139         }
140         for i, c := range p.Cmds {

```

```

141         if i > 0 {
142             s += " | "
143         }
144         s += c.String()
145     }
146     return s
147 }
148
149 func (p *PipeNode) CopyPipe() *PipeNode {
150     if p == nil {
151         return p
152     }
153     var decl []*VariableNode
154     for _, d := range p.Decl {
155         decl = append(decl, d.Copy().(*VariableNode))
156     }
157     n := newPipeline(p.Line, decl)
158     for _, c := range p.Cmds {
159         n.append(c.Copy().(*CommandNode))
160     }
161     return n
162 }
163
164 func (p *PipeNode) Copy() Node {
165     return p.CopyPipe()
166 }
167
168 // ActionNode holds an action (something bounded by delimit
169 // Control actions have their own nodes; ActionNode represen
170 // ones such as field evaluations.
171 type ActionNode struct {
172     NodeType
173     Line int // The line number in the input.
174     Pipe *PipeNode // The pipeline in the action.
175 }
176
177 func newAction(line int, pipe *PipeNode) *ActionNode {
178     return &ActionNode{NodeType: NodeAction, Line: line,
179 }
180
181 func (a *ActionNode) String() string {
182     return fmt.Sprintf("{%s}", a.Pipe)
183 }
184 }
185
186 func (a *ActionNode) Copy() Node {
187     return newAction(a.Line, a.Pipe.CopyPipe())
188 }
189 }

```

```

190
191 // CommandNode holds a command (a pipeline inside an evaluat
192 type CommandNode struct {
193     NodeType
194     Args []Node // Arguments in lexical order: Identifie
195 }
196
197 func newCommand() *CommandNode {
198     return &CommandNode{NodeType: NodeCommand}
199 }
200
201 func (c *CommandNode) append(arg Node) {
202     c.Args = append(c.Args, arg)
203 }
204
205 func (c *CommandNode) String() string {
206     s := ""
207     for i, arg := range c.Args {
208         if i > 0 {
209             s += " "
210         }
211         s += arg.String()
212     }
213     return s
214 }
215
216 func (c *CommandNode) Copy() Node {
217     if c == nil {
218         return c
219     }
220     n := newCommand()
221     for _, c := range c.Args {
222         n.append(c.Copy())
223     }
224     return n
225 }
226
227 // IdentifierNode holds an identifier.
228 type IdentifierNode struct {
229     NodeType
230     Ident string // The identifier's name.
231 }
232
233 // NewIdentifier returns a new IdentifierNode with the given
234 func NewIdentifier(ident string) *IdentifierNode {
235     return &IdentifierNode{NodeType: NodeIdentifier, Ide
236 }
237
238 func (i *IdentifierNode) String() string {
239     return i.Ident

```

```

240 }
241
242 func (i *IdentifierNode) Copy() Node {
243     return NewIdentifier(i.Ident)
244 }
245
246 // VariableNode holds a list of variable names. The dollar s
247 // part of the name.
248 type VariableNode struct {
249     NodeType
250     Ident []string // Variable names in lexical order.
251 }
252
253 func newVariable(ident string) *VariableNode {
254     return &VariableNode{NodeType: NodeVariable, Ident:
255 }
256
257 func (v *VariableNode) String() string {
258     s := ""
259     for i, id := range v.Ident {
260         if i > 0 {
261             s += "."
262         }
263         s += id
264     }
265     return s
266 }
267
268 func (v *VariableNode) Copy() Node {
269     return &VariableNode{NodeType: NodeVariable, Ident:
270 }
271
272 // DotNode holds the special identifier '.'. It is represent
273 type DotNode bool
274
275 func newDot() *DotNode {
276     return nil
277 }
278
279 func (d *DotNode) Type() NodeType {
280     return NodeDot
281 }
282
283 func (d *DotNode) String() string {
284     return "."
285 }
286
287 func (d *DotNode) Copy() Node {
288     return newDot()

```

```

289 }
290
291 // FieldNode holds a field (identifier starting with '.').
292 // The names may be chained ('.x.y').
293 // The period is dropped from each ident.
294 type FieldNode struct {
295     NodeType
296     Ident []string // The identifiers in lexical order.
297 }
298
299 func newField(ident string) *FieldNode {
300     return &FieldNode{NodeType: NodeField, Ident: string
301 }
302
303 func (f *FieldNode) String() string {
304     s := ""
305     for _, id := range f.Ident {
306         s += "." + id
307     }
308     return s
309 }
310
311 func (f *FieldNode) Copy() Node {
312     return &FieldNode{NodeType: NodeField, Ident: append
313 }
314
315 // BoolNode holds a boolean constant.
316 type BoolNode struct {
317     NodeType
318     True bool // The value of the boolean constant.
319 }
320
321 func newBool(true bool) *BoolNode {
322     return &BoolNode{NodeType: NodeBool, True: true}
323 }
324
325 func (b *BoolNode) String() string {
326     if b.True {
327         return "true"
328     }
329     return "false"
330 }
331
332 func (b *BoolNode) Copy() Node {
333     return newBool(b.True)
334 }
335
336 // NumberNode holds a number: signed or unsigned integer, fl
337 // The value is parsed and stored under all the types that c

```

```

338 // This simulates in a small amount of code the behavior of
339 type NumberNode struct {
340     NodeType
341     IsInt      bool        // Number has an integral valu
342     IsUint     bool        // Number has an unsigned inte
343     IsFloat    bool        // Number has a floating-point
344     IsComplex  bool        // Number is complex.
345     Int64      int64       // The signed integer value.
346     Uint64     uint64      // The unsigned integer value.
347     Float64    float64     // The floating-point value.
348     Complex128 complex128 // The complex value.
349     Text       string      // The original textual repres
350 }
351
352 func newNumber(text string, typ itemType) (*NumberNode, error) {
353     n := &NumberNode{NodeType: NodeNumber, Text: text}
354     switch typ {
355     case itemCharConstant:
356         rune, _, tail, err := strconv.UnquoteChar(text)
357         if err != nil {
358             return nil, err
359         }
360         if tail != "" {
361             return nil, fmt.Errorf("malformed ch
362         }
363         n.Int64 = int64(rune)
364         n.IsInt = true
365         n.Uint64 = uint64(rune)
366         n.IsUint = true
367         n.Float64 = float64(rune) // odd but those a
368         n.IsFloat = true
369         return n, nil
370     case itemComplex:
371         // fmt.Sscan can parse the pair, so let it d
372         if _, err := fmt.Sscan(text, &n.Complex128);
373             err != nil {
374             return nil, err
375         }
376         n.IsComplex = true
377         n.simplifyComplex()
378         return n, nil
379     }
380     // Imaginary constants can only be complex unless th
381     if len(text) > 0 && text[len(text)-1] == 'i' {
382         f, err := strconv.ParseFloat(text[:len(text)
383         if err == nil {
384             n.IsComplex = true
385             n.Complex128 = complex(0, f)
386             n.simplifyComplex()
387             return n, nil
388         }

```

```

388     }
389     // Do integer test first so we get 0x123 etc.
390     u, err := strconv.ParseUint(text, 0, 64) // will fail
391     if err == nil {
392         n.IsUint = true
393         n.Uint64 = u
394     }
395     i, err := strconv.ParseInt(text, 0, 64)
396     if err == nil {
397         n.IsInt = true
398         n.Int64 = i
399         if i == 0 {
400             n.IsUint = true // in case of -0.
401             n.Uint64 = u
402         }
403     }
404     // If an integer extraction succeeded, promote the float
405     if n.IsInt {
406         n.IsFloat = true
407         n.Float64 = float64(n.Int64)
408     } else if n.IsUint {
409         n.IsFloat = true
410         n.Float64 = float64(n.Uint64)
411     } else {
412         f, err := strconv.ParseFloat(text, 64)
413         if err == nil {
414             n.IsFloat = true
415             n.Float64 = f
416             // If a floating-point extraction succeeded
417             if !n.IsInt && float64(int64(f)) == f {
418                 n.IsInt = true
419                 n.Int64 = int64(f)
420             }
421             if !n.IsUint && float64(uint64(f)) == f {
422                 n.IsUint = true
423                 n.Uint64 = uint64(f)
424             }
425         }
426     }
427     if !n.IsInt && !n.IsUint && !n.IsFloat {
428         return nil, fmt.Errorf("illegal number syntax")
429     }
430     return n, nil
431 }
432
433 // simplifyComplex pulls out any other types that are represented
434 // These all require that the imaginary part be zero.
435 func (n *NumberNode) simplifyComplex() {
436     n.IsFloat = imag(n.Complex128) == 0

```

```

437         if n.IsFloat {
438             n.Float64 = real(n.Complex128)
439             n.IsInt = float64(int64(n.Float64)) == n.Flo
440             if n.IsInt {
441                 n.Int64 = int64(n.Float64)
442             }
443             n.IsUint = float64(uint64(n.Float64)) == n.F
444             if n.IsUint {
445                 n.Uint64 = uint64(n.Float64)
446             }
447         }
448     }
449
450     func (n *NumberNode) String() string {
451         return n.Text
452     }
453
454     func (n *NumberNode) Copy() Node {
455         nn := new(NumberNode)
456         *nn = *n // Easy, fast, correct.
457         return nn
458     }
459
460     // StringNode holds a string constant. The value has been "u
461     type StringNode struct {
462         NodeType
463         Quoted string // The original text of the string, wi
464         Text   string // The string, after quote processing.
465     }
466
467     func newString(orig, text string) *StringNode {
468         return &StringNode{NodeType: NodeString, Quoted: ori
469     }
470
471     func (s *StringNode) String() string {
472         return s.Quoted
473     }
474
475     func (s *StringNode) Copy() Node {
476         return newString(s.Quoted, s.Text)
477     }
478
479     // endNode represents an {{end}} action. It is represented b
480     // It does not appear in the final parse tree.
481     type endNode bool
482
483     func newEnd() *endNode {
484         return nil
485     }

```

```

486
487 func (e *endNode) Type() NodeType {
488     return nodeEnd
489 }
490
491 func (e *endNode) String() string {
492     return "{{end}}"
493 }
494
495 func (e *endNode) Copy() Node {
496     return newEnd()
497 }
498
499 // elseNode represents an {{else}} action. Does not appear i
500 type elseNode struct {
501     NodeType
502     Line int // The line number in the input.
503 }
504
505 func newElse(line int) *elseNode {
506     return &elseNode{NodeType: nodeElse, Line: line}
507 }
508
509 func (e *elseNode) Type() NodeType {
510     return nodeElse
511 }
512
513 func (e *elseNode) String() string {
514     return "{{else}}"
515 }
516
517 func (e *elseNode) Copy() Node {
518     return newElse(e.Line)
519 }
520
521 // BranchNode is the common representation of if, range, and
522 type BranchNode struct {
523     NodeType
524     Line    int // The line number in the input.
525     Pipe    *PipeNode // The pipeline to be evaluated.
526     List    *ListNode // What to execute if the value i
527     ElseList *ListNode // What to execute if the value i
528 }
529
530 func (b *BranchNode) String() string {
531     name := ""
532     switch b.NodeType {
533     case NodeIf:
534         name = "if"
535     case NodeRange:

```

```

536         name = "range"
537     case NodeWith:
538         name = "with"
539     default:
540         panic("unknown branch type")
541     }
542     if b.ElseList != nil {
543         return fmt.Sprintf("{{%s %s}}%s{{else}}%s{{e
544     }
545     return fmt.Sprintf("{{%s %s}}%s{{end}}", name, b.Pip
546 }
547
548 // IfNode represents an {{if}} action and its commands.
549 type IfNode struct {
550     BranchNode
551 }
552
553 func newIf(line int, pipe *PipeNode, list, elseList *ListNod
554     return &IfNode{BranchNode{NodeType: NodeIf, Line: li
555 }
556
557 func (i *IfNode) Copy() Node {
558     return newIf(i.Line, i.Pipe.CopyPipe(), i.List.CopyL
559 }
560
561 // RangeNode represents a {{range}} action and its commands.
562 type RangeNode struct {
563     BranchNode
564 }
565
566 func newRange(line int, pipe *PipeNode, list, elseList *List
567     return &RangeNode{BranchNode{NodeType: NodeRange, Li
568 }
569
570 func (r *RangeNode) Copy() Node {
571     return newRange(r.Line, r.Pipe.CopyPipe(), r.List.Co
572 }
573
574 // WithNode represents a {{with}} action and its commands.
575 type WithNode struct {
576     BranchNode
577 }
578
579 func newWith(line int, pipe *PipeNode, list, elseList *ListN
580     return &WithNode{BranchNode{NodeType: NodeWith, Line
581 }
582
583 func (w *WithNode) Copy() Node {
584     return newWith(w.Line, w.Pipe.CopyPipe(), w.List.Cop

```

```

585 }
586
587 // TemplateNode represents a {{template}} action.
588 type TemplateNode struct {
589     NodeType
590     Line int        // The line number in the input.
591     Name string    // The name of the template (unquoted
592     Pipe *PipeNode // The command to evaluate as dot for
593 }
594
595 func newTemplate(line int, name string, pipe *PipeNode) *Tem
596     return &TemplateNode{NodeType: NodeTemplate, Line: l
597 }
598
599 func (t *TemplateNode) String() string {
600     if t.Pipe == nil {
601         return fmt.Sprintf("{{template %q}}", t.Name
602     }
603     return fmt.Sprintf("{{template %q %s}}", t.Name, t.P
604 }
605
606 func (t *TemplateNode) Copy() Node {
607     return newTemplate(t.Line, t.Name, t.Pipe.CopyPipe()
608 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/text/template/parse/parse.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package parse builds parse trees for templates as defined
6 // and html/template. Clients should use those packages to c
7 // rather than this one, which provides shared internal data
8 // intended for general use.
9 package parse
10
11 import (
12     "bytes"
13     "fmt"
14     "runtime"
15     "strconv"
16     "unicode"
17 )
18
19 // Tree is the representation of a single parsed template.
20 type Tree struct {
21     Name string // name of the template represented b
22     Root *ListNode // top-level root of the tree.
23     // Parsing only; cleared after parse.
24     funcs []map[string]interface{}
25     lex *lexer
26     token [2]item // two-token lookahead for parser.
27     peekCount int
28     vars []string // variables defined at the momen
29 }
30
31 // Parse returns a map from template name to parse.Tree, cre
32 // templates described in the argument string. The top-level
33 // given the specified name. If an error is encountered, par
34 // empty map is returned with the error.
35 func Parse(name, text, leftDelim, rightDelim string, funcs .
36     treeSet = make(map[string]*Tree)
37     _, err = New(name).Parse(text, leftDelim, rightDelim
38     return
39 }
40
41 // next returns the next token.
```

```

42 func (t *Tree) next() item {
43     if t.peekCount > 0 {
44         t.peekCount--
45     } else {
46         t.token[0] = t.lex.nextItem()
47     }
48     return t.token[t.peekCount]
49 }
50
51 // backup backs the input stream up one token.
52 func (t *Tree) backup() {
53     t.peekCount++
54 }
55
56 // backup2 backs the input stream up two tokens
57 func (t *Tree) backup2(t1 item) {
58     t.token[1] = t1
59     t.peekCount = 2
60 }
61
62 // peek returns but does not consume the next token.
63 func (t *Tree) peek() item {
64     if t.peekCount > 0 {
65         return t.token[t.peekCount-1]
66     }
67     t.peekCount = 1
68     t.token[0] = t.lex.nextItem()
69     return t.token[0]
70 }
71
72 // Parsing.
73
74 // New allocates a new parse tree with the given name.
75 func New(name string, funcs ...map[string]interface{}) *Tree
76     return &Tree{
77         Name: name,
78         funcs: funcs,
79     }
80 }
81
82 // errorf formats the error and terminates processing.
83 func (t *Tree) errorf(format string, args ...interface{}) {
84     t.Root = nil
85     format = fmt.Sprintf("template: %s:%d: %s", t.Name,
86         panic(fmt.Errorf(format, args...))
87 }
88
89 // error terminates processing.
90 func (t *Tree) error(err error) {
91     t.errorf("%s", err)

```

```

92 }
93
94 // expect consumes the next token and guarantees it has the
95 func (t *Tree) expect(expected itemType, context string) ite
96     token := t.next()
97     if token.typ != expected {
98         t.errorf("expected %s in %s; got %s", expect
99     }
100     return token
101 }
102
103 // expectEither consumes the next token and guarantees it ha
104 func (t *Tree) expectOneOf(expected1, expected2 itemType, co
105     token := t.next()
106     if token.typ != expected1 && token.typ != expected2
107         t.errorf("expected %s or %s in %s; got %s",
108     }
109     return token
110 }
111
112 // unexpected complains about the token and terminates proce
113 func (t *Tree) unexpected(token item, context string) {
114     t.errorf("unexpected %s in %s", token, context)
115 }
116
117 // recover is the handler that turns panics into returns fro
118 func (t *Tree) recover(errp *error) {
119     e := recover()
120     if e != nil {
121         if _, ok := e.(runtime.Error); ok {
122             panic(e)
123         }
124         if t != nil {
125             t.stopParse()
126         }
127         *errp = e.(error)
128     }
129     return
130 }
131
132 // startParse initializes the parser, using the lexer.
133 func (t *Tree) startParse(funcs []map[string]interface{}, le
134     t.Root = nil
135     t.lex = lex
136     t.vars = []string{"$"}
137     t.funcs = funcs
138 }
139
140 // stopParse terminates parsing.

```

```

141 func (t *Tree) stopParse() {
142     t.lex = nil
143     t.vars = nil
144     t.funcs = nil
145 }
146
147 // atEOF returns true if, possibly after spaces, we're at E0
148 func (t *Tree) atEOF() bool {
149     for {
150         token := t.peek()
151         switch token.typ {
152         case itemEOF:
153             return true
154         case itemText:
155             for _, r := range token.val {
156                 if !unicode.IsSpace(r) {
157                     return false
158                 }
159             }
160             t.next() // skip spaces.
161             continue
162         }
163         break
164     }
165     return false
166 }
167
168 // Parse parses the template definition string to construct
169 // the template for execution. If either action delimiter st
170 // default ("{{" or "}}") is used. Embedded template definit
171 // the treeSet map.
172 func (t *Tree) Parse(s, leftDelim, rightDelim string, treeSe
173     defer t.recover(&err)
174     t.startParse(funcs, lex(t.Name, s, leftDelim, rightD
175     t.parse(treeSet)
176     t.add(treeSet)
177     t.stopParse()
178     return t, nil
179 }
180
181 // add adds tree to the treeSet.
182 func (t *Tree) add(treeSet map[string]*Tree) {
183     tree := treeSet[t.Name]
184     if tree == nil || IsEmptyTree(tree.Root) {
185         treeSet[t.Name] = t
186         return
187     }
188     if !IsEmptyTree(t.Root) {
189         t.errorf("template: multiple definition of t

```

```

190     }
191 }
192
193 // isEmptyTree reports whether this tree (node) is empty of
194 func isEmptyTree(n Node) bool {
195     switch n := n.(type) {
196     case nil:
197         return true
198     case *ActionNode:
199     case *IfNode:
200     case *ListNode:
201         for _, node := range n.Nodes {
202             if !isEmptyTree(node) {
203                 return false
204             }
205         }
206         return true
207     case *RangeNode:
208     case *TemplateNode:
209     case *TextNode:
210         return len(bytes.TrimSpace(n.Text)) == 0
211     case *WithNode:
212     default:
213         panic("unknown node: " + n.String())
214     }
215     return false
216 }
217
218 // parse is the top-level parser for a template, essentially
219 // as itemList except it also parses {{define}} actions.
220 // It runs to EOF.
221 func (t *Tree) parse(treeSet map[string]*Tree) (next Node) {
222     t.Root = newList()
223     for t.peek().typ != itemEOF {
224         if t.peek().typ == itemLeftDelim {
225             delim := t.next()
226             if t.next().typ == itemDefine {
227                 newT := New("definition") //
228                 newT.startParse(t.funcs, t.l
229                 newT.parseDefinition(treeSet
230                 continue
231             }
232             t.backup2(delim)
233         }
234         n := t.textOrAction()
235         if n.Type() == nodeEnd {
236             t.errorf("unexpected %s", n)
237         }
238         t.Root.append(n)
239     }

```

```

240         return nil
241     }
242
243     // parseDefinition parses a {{define}} ... {{end}} template
244     // installs the definition in the treeSet map. The "define"
245     // been scanned.
246     func (t *Tree) parseDefinition(treeSet map[string]*Tree) {
247         const context = "define clause"
248         name := t.expectOneOf(itemString, itemRawString, con
249         var err error
250         t.Name, err = strconv.Unquote(name.val)
251         if err != nil {
252             t.error(err)
253         }
254         t.expect(itemRightDelim, context)
255         var end Node
256         t.Root, end = t.itemList()
257         if end.Type() != nodeEnd {
258             t.errorf("unexpected %s in %s", end, context
259         }
260         t.stopParse()
261         t.add(treeSet)
262     }
263
264     // itemList:
265     //     textOrAction*
266     // Terminates at {{end}} or {{else}}, returned separately.
267     func (t *Tree) itemList() (list *ListNode, next Node) {
268         list = newList()
269         for t.peek().typ != itemEOF {
270             n := t.textOrAction()
271             switch n.Type() {
272                 case nodeEnd, nodeElse:
273                     return list, n
274             }
275             list.append(n)
276         }
277         t.errorf("unexpected EOF")
278         return
279     }
280
281     // textOrAction:
282     //     text | action
283     func (t *Tree) textOrAction() Node {
284         switch token := t.next(); token.typ {
285             case itemText:
286                 return newText(token.val)
287             case itemLeftDelim:
288                 return t.action()

```

```

289         default:
290             t.unexpected(token, "input")
291     }
292     return nil
293 }
294
295 // Action:
296 //     control
297 //     command ("|" command)*
298 // Left delim is past. Now get actions.
299 // First word could be a keyword such as range.
300 func (t *Tree) action() (n Node) {
301     switch token := t.next(); token.typ {
302     case itemElse:
303         return t.elseControl()
304     case itemEnd:
305         return t.endControl()
306     case itemIf:
307         return t.ifControl()
308     case itemRange:
309         return t.rangeControl()
310     case itemTemplate:
311         return t.templateControl()
312     case itemWith:
313         return t.withControl()
314     }
315     t.backup()
316     // Do not pop variables; they persist until "end".
317     return newAction(t.lex.lineNumber(), t.pipeline("com
318 }
319
320 // Pipeline:
321 //     field or command
322 //     pipeline "|" pipeline
323 func (t *Tree) pipeline(context string) (pipe *PipeNode) {
324     var decl []*VariableNode
325     // Are there declarations?
326     for {
327         if v := t.peek(); v.typ == itemVariable {
328             t.next()
329             if next := t.peek(); next.typ == ite
330                 t.next()
331                 variable := newVariable(v.va
332                 if len(variable.Ident) != 1
333                     t.errorf("illegal va
334             }
335             decl = append(decl, variable
336             t.vars = append(t.vars, v.va
337             if next.typ == itemChar && n

```

```

338                                     if context == "range
339                                     continue
340                                     }
341                                     t.errorf("too many d
342                                     }
343                                     } else {
344                                     t.backup2(v)
345                                     }
346                                     }
347                                     break
348                                 }
349     pipe = newPipeline(t.lex.lineNumber(), decl)
350     for {
351         switch token := t.next(); token.typ {
352         case itemRightDelim:
353             if len(pipe.Cmds) == 0 {
354                 t.errorf("missing value for
355             }
356             return
357         case itemBool, itemCharConstant, itemComplex
358             itemVariable, itemNumber, itemRawStr
359             t.backup()
360             pipe.append(t.command())
361         default:
362             t.unexpected(token, context)
363         }
364     }
365     return
366 }
367
368 func (t *Tree) parseControl(context string) (lineNum int, pi
369     lineNum = t.lex.lineNumber()
370     defer t.popVars(len(t.vars))
371     pipe = t.pipeline(context)
372     var next Node
373     list, next = t.itemList()
374     switch next.Type() {
375     case nodeEnd: //done
376     case nodeElse:
377         elseList, next = t.itemList()
378         if next.Type() != nodeEnd {
379             t.errorf("expected end; found %s", n
380         }
381         elseList = elseList
382     }
383     return lineNum, pipe, list, elseList
384 }
385
386 // If:
387 //     {{if pipeline}} itemList {{end}}

```

```

388 //      {{if pipeline}} itemList {{else}} itemList {{end}}
389 // If keyword is past.
390 func (t *Tree) ifControl() Node {
391     return newIf(t.parseControl("if"))
392 }
393
394 // Range:
395 //      {{range pipeline}} itemList {{end}}
396 //      {{range pipeline}} itemList {{else}} itemList {{end}}
397 // Range keyword is past.
398 func (t *Tree) rangeControl() Node {
399     return newRange(t.parseControl("range"))
400 }
401
402 // With:
403 //      {{with pipeline}} itemList {{end}}
404 //      {{with pipeline}} itemList {{else}} itemList {{end}}
405 // If keyword is past.
406 func (t *Tree) withControl() Node {
407     return newWith(t.parseControl("with"))
408 }
409
410 // End:
411 //      {{end}}
412 // End keyword is past.
413 func (t *Tree) endControl() Node {
414     t.expect(itemRightDelim, "end")
415     return newEnd()
416 }
417
418 // Else:
419 //      {{else}}
420 // Else keyword is past.
421 func (t *Tree) elseControl() Node {
422     t.expect(itemRightDelim, "else")
423     return newElse(t.lex.lineNumber())
424 }
425
426 // Template:
427 //      {{template stringValue pipeline}}
428 // Template keyword is past. The name must be something tha
429 // to a string.
430 func (t *Tree) templateControl() Node {
431     var name string
432     switch token := t.next(); token.typ {
433     case itemString, itemRawString:
434         s, err := strconv.Unquote(token.val)
435         if err != nil {
436             t.error(err)

```

```

437         }
438         name = s
439     default:
440         t.unexpected(token, "template invocation")
441     }
442     var pipe *PipeNode
443     if t.next().typ != itemRightDelim {
444         t.backup()
445         // Do not pop variables; they persist until
446         pipe = t.pipeline("template")
447     }
448     return newTemplate(t.lex.lineNumber(), name, pipe)
449 }
450
451 // command:
452 // space-separated arguments up to a pipeline character or r
453 // we consume the pipe character but leave the right delim t
454 func (t *Tree) command() *CommandNode {
455     cmd := newCommand()
456     Loop:
457         for {
458             switch token := t.next(); token.typ {
459             case itemRightDelim:
460                 t.backup()
461                 break Loop
462             case itemPipe:
463                 break Loop
464             case itemError:
465                 t.errorf("%s", token.val)
466             case itemIdentifier:
467                 if !t.hasFunction(token.val) {
468                     t.errorf("function %q not de
469                 }
470                 cmd.append(NewIdentifier(token.val))
471             case itemDot:
472                 cmd.append(newDot())
473             case itemVariable:
474                 cmd.append(t.useVar(token.val))
475             case itemField:
476                 cmd.append(newField(token.val))
477             case itemBool:
478                 cmd.append(newBool(token.val == "tru
479             case itemCharConstant, itemComplex, itemNumb
480                 number, err := newNumber(token.val,
481                 if err != nil {
482                     t.error(err)
483                 }
484                 cmd.append(number)
485             case itemString, itemRawString:

```

```

486             s, err := strconv.Unquote(token.val)
487             if err != nil {
488                 t.error(err)
489             }
490             cmd.append(newString(token.val, s))
491         default:
492             t.unexpected(token, "command")
493         }
494     }
495     if len(cmd.Args) == 0 {
496         t.errorf("empty command")
497     }
498     return cmd
499 }
500
501 // hasFunction reports if a function name exists in the Tree
502 func (t *Tree) hasFunction(name string) bool {
503     for _, funcMap := range t.funcs {
504         if funcMap == nil {
505             continue
506         }
507         if funcMap[name] != nil {
508             return true
509         }
510     }
511     return false
512 }
513
514 // popVars trims the variable list to the specified length
515 func (t *Tree) popVars(n int) {
516     t.vars = t.vars[:n]
517 }
518
519 // useVar returns a node for a variable reference. It errors
520 // variable is not defined.
521 func (t *Tree) useVar(name string) Node {
522     v := newVariable(name)
523     for _, varName := range t.vars {
524         if varName == v.Ident[0] {
525             return v
526         }
527     }
528     t.errorf("undefined variable %q", v.Ident[0])
529     return nil
530 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/time/format.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package time
6
7 import "errors"
8
9 // These are predefined layouts for use in Time.Format.
10 // The standard time used in the layouts is:
11 //     Mon Jan 2 15:04:05 MST 2006
12 // which is Unix time 1136243045. Since MST is GMT-0700,
13 // the standard time can be thought of as
14 //     01/02 03:04:05PM '06 -0700
15 // To define your own format, write down what the standard t
16 // like formatted your way; see the values of constants like
17 // StampMicro or Kitchen for examples.
18 //
19 // Within the format string, an underscore _ represents a sp
20 // replaced by a digit if the following number (a day) has t
21 // compatibility with fixed-width Unix time formats.
22 //
23 // A decimal point followed by one or more zeros represents
24 // second, printed to the given number of decimal places. A
25 // followed by one or more nines represents a fractional sec
26 // the given number of decimal places, with trailing zeros r
27 // When parsing (only), the input may contain a fractional s
28 // field immediately after the seconds field, even if the la
29 // signify its presence. In that case a decimal point follow
30 // series of digits is parsed as a fractional second.
31 //
32 // Numeric time zone offsets format as follows:
33 //     -0700 ±hhmm
34 //     -07:00 ±hh:mm
35 // Replacing the sign in the format with a Z triggers
36 // the ISO 8601 behavior of printing Z instead of an
37 // offset for the UTC zone. Thus:
38 //     Z0700 Z or ±hhmm
39 //     Z07:00 Z or ±hh:mm
40 const (
41     ANSIC          = "Mon Jan _2 15:04:05 2006"
42     UnixDate      = "Mon Jan _2 15:04:05 MST 2006"
43     RubyDate      = "Mon Jan 02 15:04:05 -0700 2006"
44     RFC822        = "02 Jan 06 15:04 MST"
```

```

45     RFC822Z      = "02 Jan 06 15:04 -0700" // RFC822 with
46     RFC850      = "Monday, 02-Jan-06 15:04:05 MST"
47     RFC1123     = "Mon, 02 Jan 2006 15:04:05 MST"
48     RFC1123Z    = "Mon, 02 Jan 2006 15:04:05 -0700" // R
49     RFC3339     = "2006-01-02T15:04:05Z07:00"
50     RFC3339Nano = "2006-01-02T15:04:05.999999999Z07:00"
51     Kitchen     = "3:04PM"
52     // Handy time stamps.
53     Stamp       = "Jan _2 15:04:05"
54     StampMilli  = "Jan _2 15:04:05.000"
55     StampMicro  = "Jan _2 15:04:05.000000"
56     StampNano   = "Jan _2 15:04:05.000000000"
57 )
58
59 const (
60     stdLongMonth    = "January"
61     stdMonth        = "Jan"
62     stdNumMonth     = "1"
63     stdZeroMonth    = "01"
64     stdLongWeekDay  = "Monday"
65     stdWeekDay      = "Mon"
66     stdDay          = "2"
67     stdUnderDay     = "_2"
68     stdZeroDay      = "02"
69     stdHour         = "15"
70     stdHour12       = "3"
71     stdZeroHour12   = "03"
72     stdMinute       = "4"
73     stdZeroMinute   = "04"
74     stdSecond       = "5"
75     stdZeroSecond   = "05"
76     stdLongYear     = "2006"
77     stdYear         = "06"
78     stdPM           = "PM"
79     stdpm           = "pm"
80     stdTZ           = "MST"
81     stdIS08601TZ   = "Z0700" // prints Z for UTC
82     stdIS08601ColonTZ = "Z07:00" // prints Z for UTC
83     stdNumTZ        = "-0700" // always numeric
84     stdNumShortTZ   = "-07" // always numeric
85     stdNumColonTZ   = "-07:00" // always numeric
86 )
87
88 // nextStdChunk finds the first occurrence of a std string i
89 // layout and returns the text before, the std string, and t
90 func nextStdChunk(layout string) (prefix, std, suffix string)
91     for i := 0; i < len(layout); i++ {
92         switch layout[i] {
93             case 'J': // January, Jan
94                 if len(layout) >= i+7 && layout[i:i+

```

```

95         return layout[0:i], stdLongM
96     }
97     if len(layout) >= i+3 && layout[i:i+
98         return layout[0:i], stdMonth
99     }
100
101     case 'M': // Monday, Mon, MST
102         if len(layout) >= i+6 && layout[i:i+
103             return layout[0:i], stdLongW
104         }
105         if len(layout) >= i+3 {
106             if layout[i:i+3] == stdWeekD
107                 return layout[0:i],
108             }
109             if layout[i:i+3] == stdTZ {
110                 return layout[0:i],
111             }
112         }
113
114     case '0': // 01, 02, 03, 04, 05, 06
115         if len(layout) >= i+2 && '1' <= layo
116             return layout[0:i], layout[i
117         }
118
119     case '1': // 15, 1
120         if len(layout) >= i+2 && layout[i+1]
121             return layout[0:i], stdHour,
122         }
123         return layout[0:i], stdNumMonth, lay
124
125     case '2': // 2006, 2
126         if len(layout) >= i+4 && layout[i:i+
127             return layout[0:i], stdLongY
128         }
129         return layout[0:i], stdDay, layout[i
130
131     case '_': // _2
132         if len(layout) >= i+2 && layout[i+1]
133             return layout[0:i], stdUnder
134         }
135
136     case '3', '4', '5': // 3, 4, 5
137         return layout[0:i], layout[i : i+1],
138
139     case 'P': // PM
140         if len(layout) >= i+2 && layout[i+1]
141             return layout[0:i], layout[i
142         }
143

```

```

144         case 'p': // pm
145             if len(layout) >= i+2 && layout[i+1]
146                 return layout[0:i], layout[i
147             ]
148
149         case '-': // -0700, -07:00, -07
150             if len(layout) >= i+5 && layout[i:i+
151                 return layout[0:i], layout[i
152             ]
153             if len(layout) >= i+6 && layout[i:i+
154                 return layout[0:i], layout[i
155             ]
156             if len(layout) >= i+3 && layout[i:i+
157                 return layout[0:i], layout[i
158             ]
159         case 'Z': // Z0700, Z07:00
160             if len(layout) >= i+5 && layout[i:i+
161                 return layout[0:i], layout[i
162             ]
163             if len(layout) >= i+6 && layout[i:i+
164                 return layout[0:i], layout[i
165             ]
166         case '.': // .000 or .999 - repeated digits
167             if i+1 < len(layout) && (layout[i+1]
168                 ch := layout[i+1]
169                 j := i + 1
170                 for j < len(layout) && layou
171                     j++
172                 }
173                 // String of digits must end
174                 if !isDigit(layout, j) {
175                     return layout[0:i],
176                 }
177             }
178         }
179     }
180     return layout, "", ""
181 }
182
183 var longDayNames = []string{
184     "Sunday",
185     "Monday",
186     "Tuesday",
187     "Wednesday",
188     "Thursday",
189     "Friday",
190     "Saturday",
191 }
192

```

```

193 var shortDayNames = []string{
194     "Sun",
195     "Mon",
196     "Tue",
197     "Wed",
198     "Thu",
199     "Fri",
200     "Sat",
201 }
202
203 var shortMonthNames = []string{
204     "---",
205     "Jan",
206     "Feb",
207     "Mar",
208     "Apr",
209     "May",
210     "Jun",
211     "Jul",
212     "Aug",
213     "Sep",
214     "Oct",
215     "Nov",
216     "Dec",
217 }
218
219 var longMonthNames = []string{
220     "---",
221     "January",
222     "February",
223     "March",
224     "April",
225     "May",
226     "June",
227     "July",
228     "August",
229     "September",
230     "October",
231     "November",
232     "December",
233 }
234
235 // match returns true if s1 and s2 match ignoring case.
236 // It is assumed s1 and s2 are the same length.
237 func match(s1, s2 string) bool {
238     for i := 0; i < len(s1); i++ {
239         c1 := s1[i]
240         c2 := s2[i]
241         if c1 != c2 {
242             // Switch to lower-case; 'a'-'A' is

```

```

243             c1 |= 'a' - 'A'
244             c2 |= 'a' - 'A'
245             if c1 != c2 || c1 < 'a' || c1 > 'z'
246                 return false
247             }
248         }
249     }
250     return true
251 }
252
253 func lookup(tab []string, val string) (int, string, error) {
254     for i, v := range tab {
255         if len(val) >= len(v) && match(val[0:len(v)])
256             return i, val[len(v):], nil
257     }
258 }
259 return -1, val, errBad
260 }
261
262 // Duplicates functionality in strconv, but avoids dependenc
263 func itoa(x int) string {
264     var buf [32]byte
265     n := len(buf)
266     if x == 0 {
267         return "0"
268     }
269     u := uint(x)
270     if x < 0 {
271         u = -u
272     }
273     for u > 0 {
274         n--
275         buf[n] = byte(u%10 + '0')
276         u /= 10
277     }
278     if x < 0 {
279         n--
280         buf[n] = '-'
281     }
282     return string(buf[n:])
283 }
284
285 // Never printed, just needs to be non-nil for return by ato
286 var atoiError = errors.New("time: invalid number")
287
288 // Duplicates functionality in strconv, but avoids dependenc
289 func atoi(s string) (x int, err error) {
290     neg := false
291     if s != "" && s[0] == '-' {

```

```

292         neg = true
293         s = s[1:]
294     }
295     x, rem, err := leadingInt(s)
296     if err != nil || rem != "" {
297         return 0, atoiError
298     }
299     if neg {
300         x = -x
301     }
302     return x, nil
303 }
304
305 func pad(i int, padding string) string {
306     s := itoa(i)
307     if i < 10 {
308         s = padding + s
309     }
310     return s
311 }
312
313 func zeroPad(i int) string { return pad(i, "0") }
314
315 // formatNano formats a fractional second, as nanoseconds.
316 func formatNano(nanosec, n int, trim bool) string {
317     // User might give us bad data. Make sure it's posit
318     // They'll get nonsense output but it will have the
319     s := itoa(int(uint(nanosec) % 1e9))
320     // Zero pad left without fmt.
321     if len(s) < 9 {
322         s = "000000000"[:9-len(s)] + s
323     }
324     if n > 9 {
325         n = 9
326     }
327     if trim {
328         for n > 0 && s[n-1] == '0' {
329             n--
330         }
331         if n == 0 {
332             return ""
333         }
334     }
335     return "." + s[:n]
336 }
337
338 // String returns the time formatted using the format string
339 // "2006-01-02 15:04:05.999999999 -0700 MST"
340 func (t Time) String() string {

```

```

341         return t.Format("2006-01-02 15:04:05.999999999 -0700
342     }
343
344     type buffer []byte
345
346     func (b *buffer) WriteString(s string) {
347         *b = append(*b, s...)
348     }
349
350     func (b *buffer) String() string {
351         return string([]byte(*b))
352     }
353
354     // Format returns a textual representation of the time value
355     // according to layout. The layout defines the format by sh
356     // representation of the standard time,
357     //     Mon Jan 2 15:04:05 -0700 MST 2006
358     // which is then used to describe the time to be formatted.
359     // layouts ANSIC, UnixDate, RFC3339 and others describe stan
360     // representations. For more information about the formats a
361     // definition of the standard time, see the documentation fo
362     func (t Time) Format(layout string) string {
363         var (
364             year  int = -1
365             month Month
366             day   int
367             hour  int = -1
368             min   int
369             sec   int
370             b     buffer
371         )
372         // Each iteration generates one std value.
373         for {
374             prefix, std, suffix := nextStdChunk(layout)
375             b.WriteString(prefix)
376             if std == "" {
377                 break
378             }
379
380             // Compute year, month, day if needed.
381             if year < 0 {
382                 // Jan 01 02 2006
383                 if a, z := std[0], std[len(std)-1];
384                     year, month, day = t.Date()
385             }
386         }
387
388         // Compute hour, minute, second if needed.
389         if hour < 0 {
390             // 03 04 05 15 pm

```

```

391         if z := std[len(std)-1]; z == '3' ||
392             hour, min, sec = t.Clock()
393     }
394 }
395
396 var p string
397 switch std {
398 case stdYear:
399     p = zeroPad(year % 100)
400 case stdLongYear:
401     // Pad year to at least 4 digits.
402     p = itoa(year)
403     switch {
404     case year <= -1000:
405         // ok
406     case year <= -100:
407         p = p[:1] + "0" + p[1:]
408     case year <= -10:
409         p = p[:1] + "00" + p[1:]
410     case year < 0:
411         p = p[:1] + "000" + p[1:]
412     case year < 10:
413         p = "000" + p
414     case year < 100:
415         p = "00" + p
416     case year < 1000:
417         p = "0" + p
418     }
419 case stdMonth:
420     p = month.String()[:3]
421 case stdLongMonth:
422     p = month.String()
423 case stdNumMonth:
424     p = itoa(int(month))
425 case stdZeroMonth:
426     p = zeroPad(int(month))
427 case stdWeekDay:
428     p = t.Weekday().String()[:3]
429 case stdLongWeekDay:
430     p = t.Weekday().String()
431 case stdDay:
432     p = itoa(day)
433 case stdUnderDay:
434     p = pad(day, " ")
435 case stdZeroDay:
436     p = zeroPad(day)
437 case stdHour:
438     p = zeroPad(hour)
439 case stdHour12:

```

```

440         // Noon is 12PM, midnight is 12AM.
441         hr := hour % 12
442         if hr == 0 {
443             hr = 12
444         }
445         p = itoa(hr)
446     case stdZeroHour12:
447         // Noon is 12PM, midnight is 12AM.
448         hr := hour % 12
449         if hr == 0 {
450             hr = 12
451         }
452         p = zeroPad(hr)
453     case stdMinute:
454         p = itoa(min)
455     case stdZeroMinute:
456         p = zeroPad(min)
457     case stdSecond:
458         p = itoa(sec)
459     case stdZeroSecond:
460         p = zeroPad(sec)
461     case stdPM:
462         if hour >= 12 {
463             p = "PM"
464         } else {
465             p = "AM"
466         }
467     case stdpm:
468         if hour >= 12 {
469             p = "pm"
470         } else {
471             p = "am"
472         }
473     case stdISO8601TZ, stdISO8601ColonTZ, stdNum
474         // Ugly special case. We cheat and
475         // to mean "the time zone as formatt
476         -, offset := t.Zone()
477         if offset == 0 && std[0] == 'Z' {
478             p = "Z"
479             break
480         }
481         zone := offset / 60 // convert to mi
482         if zone < 0 {
483             p = "-"
484             zone = -zone
485         } else {
486             p = "+"
487         }
488         p += zeroPad(zone / 60)

```

```

489             if std == stdISO8601ColonTZ || std =
490                 p += ":"
491         }
492         p += zeroPad(zone % 60)
493     case stdTZ:
494         name, offset := t.Zone()
495         if name != "" {
496             p = name
497         } else {
498             // No time zone known for th
499             // Use the -0700 format.
500             zone := offset / 60 // conve
501             if zone < 0 {
502                 p = "-"
503                 zone = -zone
504             } else {
505                 p = "+"
506             }
507             p += zeroPad(zone / 60)
508             p += zeroPad(zone % 60)
509         }
510     default:
511         if len(std) >= 2 && (std[0:2] == ".0
512             p = formatNano(t.Nanosecond(
513         )
514     }
515     b.WriteString(p)
516     layout = suffix
517 }
518 return b.String()
519 }
520
521 var errBad = errors.New("bad value for field") // placeholde
522
523 // ParseError describes a problem parsing a time string.
524 type ParseError struct {
525     Layout      string
526     Value       string
527     LayoutElem  string
528     ValueElem   string
529     Message     string
530 }
531
532 func quote(s string) string {
533     return "\"" + s + "\""
534 }
535
536 // Error returns the string representation of a ParseError.
537 func (e *ParseError) Error() string {
538     if e.Message == "" {

```

```

539         return "parsing time " +
540             quote(e.Value) + " as " +
541             quote(e.Layout) + ": cannot parse "
542             quote(e.ValueElem) + " as " +
543             quote(e.LayoutElem)
544     }
545     return "parsing time " +
546         quote(e.Value) + e.Message
547 }
548
549 // isDigit returns true if s[i] is a decimal digit, false if
550 // if s[i] is out of range.
551 func isDigit(s string, i int) bool {
552     if len(s) <= i {
553         return false
554     }
555     c := s[i]
556     return '0' <= c && c <= '9'
557 }
558
559 // getnum parses s[0:1] or s[0:2] (fixed forces the latter)
560 // as a decimal integer and returns the integer and the
561 // remainder of the string.
562 func getnum(s string, fixed bool) (int, string, error) {
563     if !isDigit(s, 0) {
564         return 0, s, errBad
565     }
566     if !isDigit(s, 1) {
567         if fixed {
568             return 0, s, errBad
569         }
570         return int(s[0] - '0'), s[1:], nil
571     }
572     return int(s[0]-'0')*10 + int(s[1]-'0'), s[2:], nil
573 }
574
575 func cutspace(s string) string {
576     for len(s) > 0 && s[0] == ' ' {
577         s = s[1:]
578     }
579     return s
580 }
581
582 // skip removes the given prefix from value,
583 // treating runs of space characters as equivalent.
584 func skip(value, prefix string) (string, error) {
585     for len(prefix) > 0 {
586         if prefix[0] == ' ' {
587             if len(value) > 0 && value[0] != ' '

```

```

588             return "", errBad
589         }
590         prefix = cutspace(prefix)
591         value = cutspace(value)
592         continue
593     }
594     if len(value) == 0 || value[0] != prefix[0]
595         return "", errBad
596     }
597     prefix = prefix[1:]
598     value = value[1:]
599 }
600 return value, nil
601 }
602
603 // Parse parses a formatted string and returns the time value
604 // The layout defines the format by showing the representation of
605 // standard time,
606 //     Mon Jan 2 15:04:05 -0700 MST 2006
607 // which is then used to describe the string to be parsed. For
608 // ANSIC, UnixDate, RFC3339 and others describe standard representations
609 // more information about the formats and the definition of
610 // time, see the documentation for ANSIC.
611 //
612 // Elements omitted from the value are assumed to be zero or
613 // zero is impossible, one, so parsing "3:04pm" returns the
614 // corresponding to Jan 1, year 0, 15:04:00 UTC.
615 // Years must be in the range 0000..9999. The day of the week
616 // for syntax but it is otherwise ignored.
617 func Parse(layout, value string) (Time, error) {
618     alayout, avalue := layout, value
619     rangeErrString := "" // set if a value is out of range
620     amSet := false      // do we need to subtract 12 for AM
621     pmSet := false      // do we need to add 12 to the
622
623     // Time being constructed.
624     var (
625         year      int
626         month     int = 1 // January
627         day       int = 1
628         hour      int
629         min       int
630         sec       int
631         nsec      int
632         z         *Location
633         zoneOffset int = -1
634         zoneName  string
635     )
636

```

```

637 // Each iteration processes one std value.
638 for {
639     var err error
640     prefix, std, suffix := nextStdChunk(layout)
641     value, err = skip(value, prefix)
642     if err != nil {
643         return Time{}, &ParseError{alayout,
644     }
645     if len(std) == 0 {
646         if len(value) != 0 {
647             return Time{}, &ParseError{a
648         }
649         break
650     }
651     layout = suffix
652     var p string
653     switch std {
654     case stdYear:
655         if len(value) < 2 {
656             err = errBad
657             break
658         }
659         p, value = value[0:2], value[2:]
660         year, err = atoi(p)
661         if year >= 69 { // Unix time starts
662             year += 1900
663         } else {
664             year += 2000
665         }
666     case stdLongYear:
667         if len(value) < 4 || !isDigit(value,
668             err = errBad
669             break
670         }
671         p, value = value[0:4], value[4:]
672         year, err = atoi(p)
673     case stdMonth:
674         month, value, err = lookup(shortMont
675     case stdLongMonth:
676         month, value, err = lookup(longMonth
677     case stdNumMonth, stdZeroMonth:
678         month, value, err = getnum(value, st
679         if month <= 0 || 12 < month {
680             rangeErrString = "month"
681         }
682     case stdWeekDay:
683         // Ignore weekday except for error c
684         _, value, err = lookup(shortDayNames
685     case stdLongWeekDay:
686         _, value, err = lookup(longDayNames,

```

```

687     case stdDay, stdUnderDay, stdZeroDay:
688         if std == stdUnderDay && len(value)
689             value = value[1:]
690     }
691     day, value, err = getnum(value, std
692     if day < 0 || 31 < day {
693         rangeErrString = "day"
694     }
695     case stdHour:
696         hour, value, err = getnum(value, fal
697         if hour < 0 || 24 <= hour {
698             rangeErrString = "hour"
699         }
700     case stdHour12, stdZeroHour12:
701         hour, value, err = getnum(value, std
702         if hour < 0 || 12 < hour {
703             rangeErrString = "hour"
704         }
705     case stdMinute, stdZeroMinute:
706         min, value, err = getnum(value, std
707         if min < 0 || 60 <= min {
708             rangeErrString = "minute"
709         }
710     case stdSecond, stdZeroSecond:
711         sec, value, err = getnum(value, std
712         if sec < 0 || 60 <= sec {
713             rangeErrString = "second"
714         }
715         // Special case: do we have a fracti
716         // fractional second in the format?
717         if len(value) >= 2 && value[0] == '.'
718             _, std, _ := nextStdChunk(la
719             if len(std) > 0 && std[0] ==
720                 // Fractional second
721                 break
722         }
723         // No fractional second in t
724         n := 2
725         for ; n < len(value) && isDi
726         }
727         nsec, rangeErrString, err =
728         value = value[n:]
729     }
730     case stdPM:
731         if len(value) < 2 {
732             err = errBad
733             break
734         }
735         p, value = value[0:2], value[2:]

```

```

736         switch p {
737         case "PM":
738             pmSet = true
739         case "AM":
740             amSet = true
741         default:
742             err = errBad
743         }
744     case stdpm:
745         if len(value) < 2 {
746             err = errBad
747             break
748         }
749         p, value = value[0:2], value[2:]
750         switch p {
751         case "pm":
752             pmSet = true
753         case "am":
754             amSet = true
755         default:
756             err = errBad
757         }
758     case stdISO8601TZ, stdISO8601ColonTZ, stdNum
759         if std[0] == 'Z' && len(value) >= 1
760             value = value[1:]
761             z = UTC
762             break
763         }
764         var sign, hour, min string
765         if std == stdISO8601ColonTZ || std =
766             if len(value) < 6 {
767                 err = errBad
768                 break
769             }
770             if value[3] != ':' {
771                 err = errBad
772                 break
773             }
774             sign, hour, min, value = val
775     } else if std == stdNumShortTZ {
776         if len(value) < 3 {
777             err = errBad
778             break
779         }
780         sign, hour, min, value = val
781     } else {
782         if len(value) < 5 {
783             err = errBad
784             break

```

```

785         }
786         sign, hour, min, value = val
787     }
788     var hr, mm int
789     hr, err = atoi(hour)
790     if err == nil {
791         mm, err = atoi(min)
792     }
793     zoneOffset = (hr*60 + mm) * 60 // of
794     switch sign[0] {
795     case '+':
796     case '-':
797         zoneOffset = -zoneOffset
798     default:
799         err = errBad
800     }
801     case stdTZ:
802         // Does it look like a time zone?
803         if len(value) >= 3 && value[0:3] ==
804             z = UTC
805             value = value[3:]
806             break
807     }
808
809     if len(value) >= 3 && value[2] == 'T
810         p, value = value[0:3], value
811     } else if len(value) >= 4 && value[3]
812         p, value = value[0:4], value
813     } else {
814         err = errBad
815         break
816     }
817     for i := 0; i < len(p); i++ {
818         if p[i] < 'A' || 'Z' < p[i]
819             err = errBad
820     }
821     }
822     if err != nil {
823         break
824     }
825     // It's a valid format.
826     zoneName = p
827     default:
828     if len(value) < len(std) {
829         err = errBad
830         break
831     }
832     if len(std) >= 2 && std[0:2] == ".0"
833         nsec, rangeErrString, err =
834         value = value[len(std):]

```

```

835         }
836     }
837     if rangeErrString != "" {
838         return Time{}, &ParseError{alayout,
839     }
840     if err != nil {
841         return Time{}, &ParseError{alayout,
842     }
843 }
844 if pmSet && hour < 12 {
845     hour += 12
846 } else if amSet && hour == 12 {
847     hour = 0
848 }
849
850 // TODO: be more aggressive checking day?
851 if z != nil {
852     return Date(year, Month(month), day, hour, m
853 }
854
855 t := Date(year, Month(month), day, hour, min, sec, n
856 if zoneOffset != -1 {
857     t.sec -= int64(zoneOffset)
858
859     // Look for local zone with the given offset
860     // If that zone was in effect at the given t
861     name, offset, _, _, _ := Local.lookup(t.sec
862     if offset == zoneOffset && (zoneName == "" |
863         t.loc = Local
864         return t, nil
865     }
866
867     // Otherwise create fake zone to record offs
868     t.loc = FixedZone(zoneName, zoneOffset)
869     return t, nil
870 }
871
872 if zoneName != "" {
873     // Look for local zone with the given offset
874     // If that zone was in effect at the given t
875     offset, _, ok := Local.lookupName(zoneName)
876     if ok {
877         name, off, _, _, _ := Local.lookup(t
878         if name == zoneName && off == offset
879             t.sec -= int64(offset)
880             t.loc = Local
881             return t, nil
882         }
883     }

```

```

884
885         // Otherwise, create fake zone with unknown
886         t.loc = FixedZone(zoneName, 0)
887         return t, nil
888     }
889
890     // Otherwise, fall back to UTC.
891     return t, nil
892 }
893
894 func parseNanoseconds(value string, nbytes int) (ns int, ran
895     if value[0] != '.' {
896         err = errBad
897         return
898     }
899     ns, err = atoi(value[1:nbytes])
900     if err != nil {
901         return
902     }
903     if ns < 0 || 1e9 <= ns {
904         rangeErrString = "fractional second"
905         return
906     }
907     // We need nanoseconds, which means scaling by the n
908     // of missing digits in the format, maximum length 1
909     // longer than 10, we won't scale.
910     scaleDigits := 10 - nbytes
911     for i := 0; i < scaleDigits; i++ {
912         ns *= 10
913     }
914     return
915 }
916
917 var errLeadingInt = errors.New("time: bad [0-9]*") // never
918
919 // leadingInt consumes the leading [0-9]* from s.
920 func leadingInt(s string) (x int, rem string, err error) {
921     i := 0
922     for ; i < len(s); i++ {
923         c := s[i]
924         if c < '0' || c > '9' {
925             break
926         }
927         if x >= (1<<31-10)/10 {
928             // overflow
929             return 0, "", errLeadingInt
930         }
931         x = x*10 + int(c) - '0'
932     }

```

```

933         return x, s[i:], nil
934     }
935
936     var unitMap = map[string]float64{
937         "ns": float64(Nanosecond),
938         "us": float64(Microsecond),
939         "µs": float64(Microsecond), // U+00B5 = micro symbol
940         "μs": float64(Microsecond), // U+03BC = Greek letter
941         "ms": float64(Millisecond),
942         "s": float64(Second),
943         "m": float64(Minute),
944         "h": float64(Hour),
945     }
946
947     // ParseDuration parses a duration string.
948     // A duration string is a possibly signed sequence of
949     // decimal numbers, each with optional fraction and a unit s
950     // such as "300ms", "-1.5h" or "2h45m".
951     // Valid time units are "ns", "us" (or "µs"), "ms", "s", "m"
952     func ParseDuration(s string) (Duration, error) {
953         // [-+]?([0-9]*(\.[0-9]*)?[a-z]+)+
954         orig := s
955         f := float64(0)
956         neg := false
957
958         // Consume [-+]?
959         if s != "" {
960             c := s[0]
961             if c == '-' || c == '+' {
962                 neg = c == '-'
963                 s = s[1:]
964             }
965         }
966         // Special case: if all that is left is "0", this is
967         if s == "0" {
968             return 0, nil
969         }
970         if s == "" {
971             return 0, errors.New("time: invalid duration")
972         }
973         for s != "" {
974             g := float64(0) // this element of the sequence
975
976             var x int
977             var err error
978
979             // The next character must be [0-9.]
980             if !(s[0] == '.' || ('0' <= s[0] && s[0] <=
981                 return 0, errors.New("time: invalid
982         }

```

```

983 // Consume [0-9]*
984 pl := len(s)
985 x, s, err = leadingInt(s)
986 if err != nil {
987     return 0, errors.New("time: invalid
988 }
989 g = float64(x)
990 pre := pl != len(s) // whether we consumed a
991
992 // Consume (\.[0-9]*)?
993 post := false
994 if s != "" && s[0] == '.' {
995     s = s[1:]
996     pl := len(s)
997     x, s, err = leadingInt(s)
998     if err != nil {
999         return 0, errors.New("time:
1000     }
1001     scale := 1
1002     for n := pl - len(s); n > 0; n-- {
1003         scale *= 10
1004     }
1005     g += float64(x) / float64(scale)
1006     post = pl != len(s)
1007 }
1008 if !pre && !post {
1009     // no digits (e.g. ".s" or "-.s")
1010     return 0, errors.New("time: invalid
1011 }
1012
1013 // Consume unit.
1014 i := 0
1015 for ; i < len(s); i++ {
1016     c := s[i]
1017     if c == '.' || ('0' <= c && c <= '9'
1018         break
1019     }
1020 }
1021 if i == 0 {
1022     return 0, errors.New("time: missing
1023 }
1024 u := s[:i]
1025 s = s[i:]
1026 unit, ok := unitMap[u]
1027 if !ok {
1028     return 0, errors.New("time: unknown
1029 }
1030
1031 f += g * unit

```

```
1032     }
1033
1034     if neg {
1035         f = -f
1036     }
1037     return Duration(f), nil
1038 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/time/sleep.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package time
6
7 // Sleep pauses the current goroutine for the duration d.
8 func Sleep(d Duration)
9
10 func nano() int64 {
11     sec, nsec := now()
12     return sec*1e9 + int64(nsec)
13 }
14
15 // Interface to timers implemented in package runtime.
16 // Must be in sync with ../runtime/runtime.h:^struct.Timer$
17 type runtimeTimer struct {
18     i      int32
19     when   int64
20     period int64
21     f      func(int64, interface{})
22     arg    interface{}
23 }
24
25 func startTimer(*runtimeTimer)
26 func stopTimer(*runtimeTimer) bool
27
28 // The Timer type represents a single event.
29 // When the Timer expires, the current time will be sent on
30 // unless the Timer was created by AfterFunc.
31 type Timer struct {
32     C <-chan Time
33     r runtimeTimer
34 }
35
36 // Stop prevents the Timer from firing.
37 // It returns true if the call stops the timer, false if the
38 // expired or stopped.
39 func (t *Timer) Stop() (ok bool) {
40     return stopTimer(&t.r)
41 }
42
43 // NewTimer creates a new Timer that will send
44 // the current time on its channel after at least duration d
```

```

45 func NewTimer(d Duration) *Timer {
46     c := make(chan Time, 1)
47     t := &Timer{
48         C: c,
49         r: runtimeTimer{
50             when: nano() + int64(d),
51             f:     sendTime,
52             arg:  c,
53         },
54     }
55     startTimer(&t.r)
56     return t
57 }
58
59 func sendTime(now int64, c interface{}) {
60     // Non-blocking send of time on c.
61     // Used in NewTimer, it cannot block anyway (buffer)
62     // Used in NewTicker, dropping sends on the floor is
63     // the desired behavior when the reader gets behind,
64     // because the sends are periodic.
65     select {
66     case c.(chan Time) <- Unix(0, now):
67     default:
68     }
69 }
70
71 // After waits for the duration to elapse and then sends the
72 // on the returned channel.
73 // It is equivalent to NewTimer(d).C.
74 func After(d Duration) <-chan Time {
75     return NewTimer(d).C
76 }
77
78 // AfterFunc waits for the duration to elapse and then calls
79 // in its own goroutine. It returns a Timer that can
80 // be used to cancel the call using its Stop method.
81 func AfterFunc(d Duration, f func()) *Timer {
82     t := &Timer{
83         r: runtimeTimer{
84             when: nano() + int64(d),
85             f:     goFunc,
86             arg:  f,
87         },
88     }
89     startTimer(&t.r)
90     return t
91 }
92
93 func goFunc(now int64, arg interface{}) {
94     go arg.(func())()

```

95 }

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/time/sys_unix.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 package time
8
9 import (
10     "errors"
11     "syscall"
12 )
13
14 // for testing: whatever interrupts a sleep
15 func interrupt() {
16     syscall.Kill(syscall.Getpid(), syscall.SIGCHLD)
17 }
18
19 // readFile reads and returns the content of the named file.
20 // It is a trivial implementation of ioutil.ReadFile, reimpl
21 // here to avoid depending on io/ioutil or os.
22 func readFile(name string) ([]byte, error) {
23     f, err := syscall.Open(name, syscall.O_RDONLY, 0)
24     if err != nil {
25         return nil, err
26     }
27     defer syscall.Close(f)
28     var (
29         buf [4096]byte
30         ret []byte
31         n    int
32     )
33     for {
34         n, err = syscall.Read(f, buf[:])
35         if n > 0 {
36             ret = append(ret, buf[:n]...)
37         }
38         if n == 0 || err != nil {
39             break
40         }
41     }
42     return ret, err
43 }
44
```

```

45 func open(name string) (uintptr, error) {
46     fd, err := syscall.Open(name, syscall.O_RDONLY, 0)
47     if err != nil {
48         return 0, err
49     }
50     return uintptr(fd), nil
51 }
52
53 func closefd(fd uintptr) {
54     syscall.Close(int(fd))
55 }
56
57 func preadn(fd uintptr, buf []byte, off int) error {
58     whence := 0
59     if off < 0 {
60         whence = 2
61     }
62     if _, err := syscall.Seek(int(fd), int64(off), whence); err != nil {
63         return err
64     }
65     for len(buf) > 0 {
66         m, err := syscall.Read(int(fd), buf)
67         if m <= 0 {
68             if err == nil {
69                 return errors.New("short read")
70             }
71             return err
72         }
73         buf = buf[m:]
74     }
75     return nil
76 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/time/tick.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package time
6
7 import "errors"
8
9 // A Ticker holds a synchronous channel that delivers `ticks
10 // at intervals.
11 type Ticker struct {
12     C <-chan Time // The channel on which the ticks are
13     r runtimeTimer
14 }
15
16 // NewTicker returns a new Ticker containing a channel that
17 // time with a period specified by the duration argument.
18 // It adjusts the intervals or drops ticks to make up for sl
19 // The duration d must be greater than zero; if not, NewTick
20 func NewTicker(d Duration) *Ticker {
21     if d <= 0 {
22         panic(errors.New("non-positive interval for
23     })
24     // Give the channel a 1-element time buffer.
25     // If the client falls behind while reading, we drop
26     // on the floor until the client catches up.
27     c := make(chan Time, 1)
28     t := &Ticker{
29         C: c,
30         r: runtimeTimer{
31             when:    nano() + int64(d),
32             period: int64(d),
33             f:       sendTime,
34             arg:     c,
35         },
36     }
37     startTimer(&t.r)
38     return t
39 }
40
41 // Stop turns off a ticker. After Stop, no more ticks will
42 func (t *Ticker) Stop() {
43     stopTimer(&t.r)
44 }
```

```
45
46 // Tick is a convenience wrapper for NewTicker providing acc
47 // channel only. Useful for clients that have no need to sh
48 func Tick(d Duration) <-chan Time {
49     if d <= 0 {
50         return nil
51     }
52     return NewTicker(d).C
53 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/time/time.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package time provides functionality for measuring and dis
6 //
7 // The calendrical calculations always assume a Gregorian ca
8 package time
9
10 import "errors"
11
12 // A Time represents an instant in time with nanosecond prec
13 //
14 // Programs using times should typically store and pass them
15 // not pointers. That is, time variables and struct fields
16 // type time.Time, not *time.Time. A Time value can be used
17 // multiple goroutines simultaneously.
18 //
19 // Time instants can be compared using the Before, After, an
20 // The Sub method subtracts two instants, producing a Durati
21 // The Add method adds a Time and a Duration, producing a Ti
22 //
23 // The zero value of type Time is January 1, year 1, 00:00:0
24 // As this time is unlikely to come up in practice, the IsZe
25 // a simple way of detecting a time that has not been initia
26 //
27 // Each Time has associated with it a Location, consulted wh
28 // presentation form of the time, such as in the Format, Hou
29 // The methods Local, UTC, and In return a Time with a speci
30 // Changing the location in this way changes only the presen
31 // change the instant in time being denoted and therefore do
32 // computations described in earlier paragraphs.
33 //
34 type Time struct {
35     // sec gives the number of seconds elapsed since
36     // January 1, year 1 00:00:00 UTC.
37     sec int64
38
39     // nsec specifies a non-negative nanosecond
40     // offset within the second named by Seconds.
41     // It must be in the range [0, 999999999].
42     nsec int32
43
44     // loc specifies the Location that should be used to
```

```

45         // determine the minute, hour, month, day, and year
46         // that correspond to this Time.
47         // Only the zero Time has a nil Location.
48         // In that case it is interpreted to mean UTC.
49         loc *Location
50     }
51
52     // After reports whether the time instant t is after u.
53     func (t Time) After(u Time) bool {
54         return t.sec > u.sec || t.sec == u.sec && t.nsec > u
55     }
56
57     // Before reports whether the time instant t is before u.
58     func (t Time) Before(u Time) bool {
59         return t.sec < u.sec || t.sec == u.sec && t.nsec < u
60     }
61
62     // Equal reports whether t and u represent the same time ins
63     // Two times can be equal even if they are in different loca
64     // For example, 6:00 +0200 CEST and 4:00 UTC are Equal.
65     // This comparison is different from using t == u, which als
66     // the locations.
67     func (t Time) Equal(u Time) bool {
68         return t.sec == u.sec && t.nsec == u.nsec
69     }
70
71     // A Month specifies a month of the year (January = 1, ...).
72     type Month int
73
74     const (
75         January Month = 1 + iota
76         February
77         March
78         April
79         May
80         June
81         July
82         August
83         September
84         October
85         November
86         December
87     )
88
89     var months = [...]string{
90         "January",
91         "February",
92         "March",
93         "April",
94         "May",

```

```

95         "June",
96         "July",
97         "August",
98         "September",
99         "October",
100        "November",
101        "December",
102    }
103
104    // String returns the English name of the month ("January",
105    func (m Month) String() string { return months[m-1] }
106
107    // A Weekday specifies a day of the week (Sunday = 0, ...).
108    type Weekday int
109
110    const (
111        Sunday Weekday = iota
112        Monday
113        Tuesday
114        Wednesday
115        Thursday
116        Friday
117        Saturday
118    )
119
120    var days = [...]string{
121        "Sunday",
122        "Monday",
123        "Tuesday",
124        "Wednesday",
125        "Thursday",
126        "Friday",
127        "Saturday",
128    }
129
130    // String returns the English name of the day ("Sunday", "Mo
131    func (d Weekday) String() string { return days[d] }
132
133    // Computations on time.
134    //
135    // The zero value for a Time is defined to be
136    //     January 1, year 1, 00:00:00.000000000 UTC
137    // which (1) looks like a zero, or as close as you can get i
138    // (1-1-1 00:00:00 UTC), (2) is unlikely enough to arise in
139    // be a suitable "not set" sentinel, unlike Jan 1 1970, and
140    // non-negative year even in time zones west of UTC, unlike
141    // 00:00:00 UTC, which would be 12-31-(-1) 19:00:00 in New Y
142    //
143    // The zero Time value does not force a specific epoch for t

```

```
144 // representation. For example, to use the Unix epoch inter
145 // could define that to distinguish a zero value from Jan 1
146 // time would be represented by sec=-1, nsec=1e9. However,
147 // suggest a representation, namely using 1-1-1 00:00:00 UTC
148 // epoch, and that's what we do.
149 //
150 // The Add and Sub computations are oblivious to the choice
151 //
152 // The presentation computations - year, month, minute, and
153 // rely heavily on division and modulus by positive constant
154 // calendrical calculations we want these divisions to round
155 // for negative values, so that the remainder is always posi
156 // Go's division (like most hardware division instructions)
157 // zero. We can still do those computations and then adjust
158 // for a negative numerator, but it's annoying to write the
159 // over and over. Instead, we can change to a different epo
160 // ago that all the times we care about will be positive, an
161 // to zero and round down coincide. These presentation rout
162 // have to add the zone offset, so adding the translation to
163 // alternate epoch is cheap. For example, having a non-nega
164 // means that we can write
165 //
166 //     sec = t % 60
167 //
168 // instead of
169 //
170 //     sec = t % 60
171 //     if sec < 0 {
172 //         sec += 60
173 //     }
174 //
175 // everywhere.
176 //
177 // The calendar runs on an exact 400 year cycle: a 400-year
178 // printed for 1970-2469 will apply as well to 2470-2869. E
179 // of the week match up. It simplifies the computations to
180 // cycle boundaries so that the exceptional years are always
181 // long as possible. That means choosing a year equal to 1
182 // that the first leap year is the 4th year, the first misse
183 // is the 100th year, and the missed missed leap year is the
184 // So we'd prefer instead to print a calendar for 2001-2400
185 // for 2401-2800.
186 //
187 // Finally, it's convenient if the delta between the Unix ep
188 // long-ago epoch is representable by an int64 constant.
189 //
190 // These three considerations—choose an epoch as early as po
191 // uses a year equal to 1 mod 400, and that is no more than
192 // earlier than 1970—bring us to the year -292277022399. We
```

```

193 // this year as the absolute zero year, and to times measure
194 // seconds since this year as absolute times.
195 //
196 // Times measured as an int64 seconds since the year 1—the r
197 // used for Time's sec field—are called internal times.
198 //
199 // Times measured as an int64 seconds since the year 1970 ar
200 // times.
201 //
202 // It is tempting to just use the year 1 as the absolute epo
203 // that the routines are only valid for years >= 1. However
204 // routines would then be invalid when displaying the epoch
205 // west of UTC, since it is year 0. It doesn't seem tenable
206 // printing the zero time correctly isn't supported in half
207 // zones. By comparison, it's reasonable to mishandle some
208 // the year -292277022399.
209 //
210 // All this is opaque to clients of the API and can be chang
211 // better implementation presents itself.
212
213 const (
214     // The unsigned zero year for internal calculations.
215     // Must be 1 mod 400, and times before it will not c
216     // but otherwise can be changed at will.
217     absoluteZeroYear = -292277022399
218
219     // The year of the zero Time.
220     // Assumed by the unixToInternal computation below.
221     internalYear = 1
222
223     // The year of the zero Unix time.
224     unixYear = 1970
225
226     // Offsets to convert between internal and absolute
227     absoluteToInternal int64 = (absoluteZeroYear - inter
228     internalToAbsolute      = -absoluteToInternal
229
230     unixToInternal int64 = (1969*365 + 1969/4 - 1969/100
231     internalToUnix int64 = -unixToInternal
232 )
233
234 // IsZero reports whether t represents the zero time instant
235 // January 1, year 1, 00:00:00 UTC.
236 func (t Time) IsZero() bool {
237     return t.sec == 0 && t.nsec == 0
238 }
239
240 // abs returns the time t as an absolute time, adjusted by t
241 // It is called when computing a presentation property like
242 func (t Time) abs() uint64 {

```

```

243         l := t.loc
244         if l == nil {
245             l = &utcLoc
246         }
247         // Avoid function call if we hit the local time cach
248         sec := t.sec + internalToUnix
249         if l != &utcLoc {
250             if l.cacheZone != nil && l.cacheStart <= sec
251                 sec += int64(l.cacheZone.offset)
252             } else {
253                 _, offset, _, _, _ := l.lookup(sec)
254                 sec += int64(offset)
255             }
256         }
257         return uint64(sec + (unixToInternal + internalToAbso
258     }
259
260 // Date returns the year, month, and day in which t occurs.
261 func (t Time) Date() (year int, month Month, day int) {
262     year, month, day, _ = t.date(true)
263     return
264 }
265
266 // Year returns the year in which t occurs.
267 func (t Time) Year() int {
268     year, _, _, _ := t.date(false)
269     return year
270 }
271
272 // Month returns the month of the year specified by t.
273 func (t Time) Month() Month {
274     _, month, _, _ := t.date(true)
275     return month
276 }
277
278 // Day returns the day of the month specified by t.
279 func (t Time) Day() int {
280     _, _, day, _ := t.date(true)
281     return day
282 }
283
284 // Weekday returns the day of the week specified by t.
285 func (t Time) Weekday() Weekday {
286     // January 1 of the absolute year, like January 1 of
287     sec := (t.abs() + uint64(Monday)*secondsPerDay) % se
288     return Weekday(int(sec) / secondsPerDay)
289 }
290
291 // ISOWeek returns the ISO 8601 year and week number in whic

```

```

292 // Week ranges from 1 to 53. Jan 01 to Jan 03 of year n might
293 // week 52 or 53 of year n-1, and Dec 29 to Dec 31 might belong
294 // of year n+1.
295 func (t Time) ISOWeek() (year, week int) {
296     year, month, day, yday := t.date(true)
297     wday := int(t.Weekday()+6) % 7 // weekday but Monday
298     const (
299         Mon int = iota
300         Tue
301         Wed
302         Thu
303         Fri
304         Sat
305         Sun
306     )
307
308     // Calculate week as number of Mondays in year up to
309     // and including today, plus 1 because the first week
310     // Putting the + 1 inside the numerator as a + 7 keeps the
311     // numerator from being negative, which would cause
312     // round incorrectly.
313     week = (yday - wday + 7) / 7
314
315     // The week number is now correct under the assumption
316     // that the first Monday of the year is in week 1.
317     // If Jan 1 is a Tuesday, Wednesday, or Thursday, then
318     // it is actually in week 2.
319     jan1wday := (wday - yday + 7*53) % 7
320     if Tue <= jan1wday && jan1wday <= Thu {
321         week++
322     }
323
324     // If the week number is still 0, we're in early Jan
325     // the last week of last year.
326     if week == 0 {
327         year--
328         week = 52
329         // A year has 53 weeks when Jan 1 or Dec 31
330         // meaning Jan 1 of the next year is a Friday
331         // or it was a leap year and Jan 1 of the next year
332         // is a Saturday
333         if jan1wday == Fri || (jan1wday == Sat && isLeapYear(year)) {
334             week++
335         }
336     }
337
338     // December 29 to 31 are in week 1 of next year if
339     // they are after the last Thursday of the year and
340     // December 31 is a Monday, Tuesday, or Wednesday.
    if month == December && day >= 29 && wday < Thu {

```

```

341         if dec31wday := (wday + 31 - day) % 7; Mon <
342             year++
343             week = 1
344     }
345 }
346
347     return
348 }
349
350 // Clock returns the hour, minute, and second within the day
351 func (t Time) Clock() (hour, min, sec int) {
352     sec = int(t.abs() % secondsPerDay)
353     hour = sec / secondsPerHour
354     sec -= hour * secondsPerHour
355     min = sec / secondsPerMinute
356     sec -= min * secondsPerMinute
357     return
358 }
359
360 // Hour returns the hour within the day specified by t, in t
361 func (t Time) Hour() int {
362     return int(t.abs()%secondsPerDay) / secondsPerHour
363 }
364
365 // Minute returns the minute offset within the hour specifie
366 func (t Time) Minute() int {
367     return int(t.abs()%secondsPerHour) / secondsPerMinut
368 }
369
370 // Second returns the second offset within the minute specif
371 func (t Time) Second() int {
372     return int(t.abs() % secondsPerMinute)
373 }
374
375 // Nanosecond returns the nanosecond offset within the secon
376 // in the range [0, 999999999].
377 func (t Time) Nanosecond() int {
378     return int(t.nsec)
379 }
380
381 // A Duration represents the elapsed time between two instan
382 // as an int64 nanosecond count. The representation limits
383 // largest representable duration to approximately 290 years
384 type Duration int64
385
386 // Common durations. There is no definition for units of Da
387 // to avoid confusion across daylight savings time zone tran
388 //
389 // To count the number of units in a Duration, divide:
390 //     second := time.Second

```

```

391 //      fmt.Print(int64(second/time.Millisecond)) // prints
392 //
393 // To convert an integer number of units to a Duration, mult
394 //      seconds := 10
395 //      fmt.Print(time.Duration(seconds)*time.Second) // pri
396 //
397 const (
398     Nanosecond  Duration = 1
399     Microsecond      = 1000 * Nanosecond
400     Millisecond     = 1000 * Microsecond
401     Second          = 1000 * Millisecond
402     Minute          = 60 * Second
403     Hour            = 60 * Minute
404 )
405
406 // String returns a string representing the duration in the
407 // Leading zero units are omitted. As a special case, durat
408 // second format use a smaller unit (milli-, micro-, or nano
409 // that the leading digit is non-zero. The zero duration fo
410 // with no unit.
411 func (d Duration) String() string {
412     // Largest time is 2540400h10m10.000000000s
413     var buf [32]byte
414     w := len(buf)
415
416     u := uint64(d)
417     neg := d < 0
418     if neg {
419         u = -u
420     }
421
422     if u < uint64(Second) {
423         // Special case: if duration is smaller than
424         // use smaller units, like 1.2ms
425         var (
426             prec int
427             unit byte
428         )
429         switch {
430         case u == 0:
431             return "0"
432         case u < uint64(Microsecond):
433             // print nanoseconds
434             prec = 0
435             unit = 'n'
436         case u < uint64(Millisecond):
437             // print microseconds
438             prec = 3
439             unit = 'u'

```

```

440         default:
441             // print milliseconds
442             prec = 6
443             unit = 'm'
444         }
445         w -= 2
446         buf[w] = unit
447         buf[w+1] = 's'
448         w, u = fmtFrac(buf[:w], u, prec)
449         w = fmtInt(buf[:w], u)
450     } else {
451         w--
452         buf[w] = 's'
453
454         w, u = fmtFrac(buf[:w], u, 9)
455
456         // u is now integer seconds
457         w = fmtInt(buf[:w], u%60)
458         u /= 60
459
460         // u is now integer minutes
461         if u > 0 {
462             w--
463             buf[w] = 'm'
464             w = fmtInt(buf[:w], u%60)
465             u /= 60
466
467             // u is now integer hours
468             // Stop at hours because days can be
469             if u > 0 {
470                 w--
471                 buf[w] = 'h'
472                 w = fmtInt(buf[:w], u)
473             }
474         }
475     }
476
477     if neg {
478         w--
479         buf[w] = '-'
480     }
481
482     return string(buf[w:])
483 }
484
485 // fmtFrac formats the fraction of v/10**prec (e.g., ".12345
486 // tail of buf, omitting trailing zeros. it omits the decim
487 // point too when the fraction is 0. It returns the index w
488 // output bytes begin and the value v/10**prec.

```

```

489 func fmtFrac(buf []byte, v uint64, prec int) (nw int, nv uin
490 // Omit trailing zeros up to and including decimal p
491 w := len(buf)
492 print := false
493 for i := 0; i < prec; i++ {
494     digit := v % 10
495     print = print || digit != 0
496     if print {
497         w--
498         buf[w] = byte(digit) + '0'
499     }
500     v /= 10
501 }
502 if print {
503     w--
504     buf[w] = '.'
505 }
506 return w, v
507 }
508
509 // fmtInt formats v into the tail of buf.
510 // It returns the index where the output begins.
511 func fmtInt(buf []byte, v uint64) int {
512     w := len(buf)
513     if v == 0 {
514         w--
515         buf[w] = '0'
516     } else {
517         for v > 0 {
518             w--
519             buf[w] = byte(v%10) + '0'
520             v /= 10
521         }
522     }
523     return w
524 }
525
526 // Nanoseconds returns the duration as an integer nanosecond
527 func (d Duration) Nanoseconds() int64 { return int64(d) }
528
529 // These methods return float64 because the dominant
530 // use case is for printing a floating point number like 1.5
531 // a truncation to integer would make them not useful in the
532 // Splitting the integer and fraction ourselves guarantees t
533 // converting the returned float64 to an integer rounds the
534 // way that a pure integer conversion would have, even in ca
535 // where, say, float64(d.Nanoseconds())/1e9 would have round
536 // differently.
537
538 // Seconds returns the duration as a floating point number o

```

```

539 func (d Duration) Seconds() float64 {
540     sec := d / Second
541     nsec := d % Second
542     return float64(sec) + float64(nsec)*1e-9
543 }
544
545 // Minutes returns the duration as a floating point number c
546 func (d Duration) Minutes() float64 {
547     min := d / Minute
548     nsec := d % Minute
549     return float64(min) + float64(nsec)*(1e-9/60)
550 }
551
552 // Hours returns the duration as a floating point number of
553 func (d Duration) Hours() float64 {
554     hour := d / Hour
555     nsec := d % Hour
556     return float64(hour) + float64(nsec)*(1e-9/60/60)
557 }
558
559 // Add returns the time t+d.
560 func (t Time) Add(d Duration) Time {
561     t.sec += int64(d / 1e9)
562     t.nsec += int32(d % 1e9)
563     if t.nsec >= 1e9 {
564         t.sec++
565         t.nsec -= 1e9
566     } else if t.nsec < 0 {
567         t.sec--
568         t.nsec += 1e9
569     }
570     return t
571 }
572
573 // Sub returns the duration t-u.
574 // To compute t-d for a duration d, use t.Add(-d).
575 func (t Time) Sub(u Time) Duration {
576     return Duration(t.sec-u.sec)*Second + Duration(t.nsec-u.nsec)
577 }
578
579 // Since returns the time elapsed since t.
580 // It is shorthand for time.Now().Sub(t).
581 func Since(t Time) Duration {
582     return Now().Sub(t)
583 }
584
585 // AddDate returns the time corresponding to adding the
586 // given number of years, months, and days to t.
587 // For example, AddDate(-1, 2, 3) applied to January 1, 2011

```

```

588 // returns March 4, 2010.
589 //
590 // AddDate normalizes its result in the same way that Date d
591 // so, for example, adding one month to October 31 yields
592 // December 1, the normalized form for November 31.
593 func (t Time) AddDate(years int, months int, days int) Time
594     year, month, day := t.Date()
595     hour, min, sec := t.Clock()
596     return Date(year+years, month+Month(months), day+day
597 }
598
599 const (
600     secondsPerMinute = 60
601     secondsPerHour   = 60 * 60
602     secondsPerDay     = 24 * secondsPerHour
603     secondsPerWeek    = 7 * secondsPerDay
604     daysPer400Years   = 365*400 + 97
605     daysPer100Years   = 365*100 + 24
606     daysPer4Years     = 365*4 + 1
607     days1970To2001    = 31*365 + 8
608 )
609
610 // date computes the year and, only when full=true,
611 // the month and day in which t occurs.
612 func (t Time) date(full bool) (year int, month Month, day in
613     // Split into time and day.
614     d := t.abs() / secondsPerDay
615
616     // Account for 400 year cycles.
617     n := d / daysPer400Years
618     y := 400 * n
619     d -= daysPer400Years * n
620
621     // Cut off 100-year cycles.
622     // The last cycle has one extra leap year, so on the
623     // of that year, day / daysPer100Years will be 4 ins
624     // Cut it back down to 3 by subtracting n>>2.
625     n = d / daysPer100Years
626     n -= n >> 2
627     y += 100 * n
628     d -= daysPer100Years * n
629
630     // Cut off 4-year cycles.
631     // The last cycle has a missing leap year, which doe
632     // affect the computation.
633     n = d / daysPer4Years
634     y += 4 * n
635     d -= daysPer4Years * n
636

```

```

637         // Cut off years within a 4-year cycle.
638         // The last year is a leap year, so on the last day
639         // day / 365 will be 4 instead of 3.  Cut it back do
640         // by subtracting n>>2.
641         n = d / 365
642         n -= n >> 2
643         y += n
644         d -= 365 * n
645
646         year = int(int64(y) + absoluteZeroYear)
647         yday = int(d)
648
649         if !full {
650             return
651         }
652
653         day = yday
654         if isLeap(year) {
655             // Leap year
656             switch {
657             case day > 31+29-1:
658                 // After leap day; pretend it wasn't
659                 day--
660             case day == 31+29-1:
661                 // Leap day.
662                 month = February
663                 day = 29
664                 return
665             }
666         }
667
668         // Estimate month on assumption that every month has
669         // The estimate may be too low by at most one month,
670         month = Month(day / 31)
671         end := int(daysBefore[month+1])
672         var begin int
673         if day >= end {
674             month++
675             begin = end
676         } else {
677             begin = int(daysBefore[month])
678         }
679
680         month++ // because January is 1
681         day = day - begin + 1
682         return
683     }
684
685     // daysBefore[m] counts the number of days in a non-leap year
686     // before month m begins.  There is an entry for m=12, count

```

```

687 // the number of days before January of next year (365).
688 var daysBefore = [...]int32{
689     0,
690     31,
691     31 + 28,
692     31 + 28 + 31,
693     31 + 28 + 31 + 30,
694     31 + 28 + 31 + 30 + 31,
695     31 + 28 + 31 + 30 + 31 + 30,
696     31 + 28 + 31 + 30 + 31 + 30 + 31,
697     31 + 28 + 31 + 30 + 31 + 30 + 31 + 31,
698     31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30,
699     31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31,
700     31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30,
701     31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 30
702 }
703
704 func daysIn(m Month, year int) int {
705     if m == February && isLeap(year) {
706         return 29
707     }
708     return int(daysBefore[m] - daysBefore[m-1])
709 }
710
711 // Provided by package runtime.
712 func now() (sec int64, nsec int32)
713
714 // Now returns the current local time.
715 func Now() Time {
716     sec, nsec := now()
717     return Time{sec + unixToInternal, nsec, Local}
718 }
719
720 // UTC returns t with the location set to UTC.
721 func (t Time) UTC() Time {
722     t.loc = UTC
723     return t
724 }
725
726 // Local returns t with the location set to local time.
727 func (t Time) Local() Time {
728     t.loc = Local
729     return t
730 }
731
732 // In returns t with the location information set to loc.
733 //
734 // In panics if loc is nil.
735 func (t Time) In(loc *Location) Time {

```



```

785         }
786         offset /= 60
787         if offset < -32768 || offset == -1 || offset
788             return nil, errors.New("Time.GobEnco
789         }
790         offsetMin = int16(offset)
791     }
792
793     enc := []byte{
794         timeGobVersion, // byte 0 : version
795         byte(t.sec >> 56), // bytes 1-8: seconds
796         byte(t.sec >> 48),
797         byte(t.sec >> 40),
798         byte(t.sec >> 32),
799         byte(t.sec >> 24),
800         byte(t.sec >> 16),
801         byte(t.sec >> 8),
802         byte(t.sec),
803         byte(t.nsec >> 24), // bytes 9-12: nanosecon
804         byte(t.nsec >> 16),
805         byte(t.nsec >> 8),
806         byte(t.nsec),
807         byte(offsetMin >> 8), // bytes 13-14: zone o
808         byte(offsetMin),
809     }
810
811     return enc, nil
812 }
813
814 // GobDecode implements the gob.GobDecoder interface.
815 func (t *Time) GobDecode(buf []byte) error {
816     if len(buf) == 0 {
817         return errors.New("Time.GobDecode: no data")
818     }
819
820     if buf[0] != timeGobVersion {
821         return errors.New("Time.GobDecode: unsupport
822     }
823
824     if len(buf) != /*version*/ 1+ /*sec*/ 8+ /*nsec*/ 4+
825         return errors.New("Time.GobDecode: invalid l
826     }
827
828     buf = buf[1:]
829     t.sec = int64(buf[7]) | int64(buf[6])<<8 | int64(buf
830         int64(buf[3])<<32 | int64(buf[2])<<40 | int6
831
832     buf = buf[8:]
833     t.nsec = int32(buf[3]) | int32(buf[2])<<8 | int32(bu
834

```

```

835     buf = buf[4:]
836     offset := int(int16(buf[1])|int16(buf[0])<<8) * 60
837
838     if offset == -1*60 {
839         t.loc = &utcLoc
840     } else if _, localoff, _, _, _ := Local.lookup(t.sec
841         t.loc = Local
842     } else {
843         t.loc = FixedZone("", offset)
844     }
845
846     return nil
847 }
848
849 // MarshalJSON implements the json.Marshaler interface.
850 // Time is formatted as RFC3339.
851 func (t Time) MarshalJSON() ([]byte, error) {
852     if y := t.Year(); y < 0 || y >= 10000 {
853         return nil, errors.New("Time.MarshalJSON: ye
854     }
855     return []byte(t.Format(`"` + RFC3339Nano + "`")), ni
856 }
857
858 // UnmarshalJSON implements the json.Unmarshaler interface.
859 // Time is expected in RFC3339 format.
860 func (t *Time) UnmarshalJSON(data []byte) (err error) {
861     // Fractional seconds are handled implicitly by Pars
862     *t, err = Parse(`"`+RFC3339+`"`, string(data))
863     return
864 }
865
866 // Unix returns the local Time corresponding to the given Un
867 // sec seconds and nsec nanoseconds since January 1, 1970 UT
868 // It is valid to pass nsec outside the range [0, 999999999]
869 func Unix(sec int64, nsec int64) Time {
870     if nsec < 0 || nsec >= 1e9 {
871         n := nsec / 1e9
872         sec += n
873         nsec -= n * 1e9
874         if nsec < 0 {
875             nsec += 1e9
876             sec--
877         }
878     }
879     return Time{sec + unixToInternal, int32(nsec), Local
880 }
881
882 func isLeap(year int) bool {
883     return year%4 == 0 && (year%100 != 0 || year%400 ==

```

```

884 }
885
886 // norm returns nhi, nlo such that
887 //     hi * base + lo == nhi * base + nlo
888 //     0 <= nlo < base
889 func norm(hi, lo, base int) (nhi, nlo int) {
890     if lo < 0 {
891         n := (-lo-1)/base + 1
892         hi -= n
893         lo += n * base
894     }
895     if lo >= base {
896         n := lo / base
897         hi += n
898         lo -= n * base
899     }
900     return hi, lo
901 }
902
903 // Date returns the Time corresponding to
904 //     yyyy-mm-dd hh:mm:ss + nsec nanoseconds
905 // in the appropriate zone for that time in the given locati
906 //
907 // The month, day, hour, min, sec, and nsec values may be ou
908 // their usual ranges and will be normalized during the conv
909 // For example, October 32 converts to November 1.
910 //
911 // A daylight savings time transition skips or repeats times
912 // For example, in the United States, March 13, 2011 2:15am
913 // while November 6, 2011 1:15am occurred twice. In such ca
914 // choice of time zone, and therefore the time, is not well-
915 // Date returns a time that is correct in one of the two zon
916 // in the transition, but it does not guarantee which.
917 //
918 // Date panics if loc is nil.
919 func Date(year int, month Month, day, hour, min, sec, nsec i
920     if loc == nil {
921         panic("time: missing Location in call to Dat
922     }
923
924     // Normalize month, overflowing into year.
925     m := int(month) - 1
926     year, m = norm(year, m, 12)
927     month = Month(m) + 1
928
929     // Normalize nsec, sec, min, hour, overflowing into
930     sec, nsec = norm(sec, nsec, 1e9)
931     min, sec = norm(min, sec, 60)
932     hour, min = norm(hour, min, 60)

```

```

933     day, hour = norm(day, hour, 24)
934
935     y := uint64(int64(year) - absoluteZeroYear)
936
937     // Compute days since the absolute epoch.
938
939     // Add in days from 400-year cycles.
940     n := y / 400
941     y -= 400 * n
942     d := daysPer400Years * n
943
944     // Add in 100-year cycles.
945     n = y / 100
946     y -= 100 * n
947     d += daysPer100Years * n
948
949     // Add in 4-year cycles.
950     n = y / 4
951     y -= 4 * n
952     d += daysPer4Years * n
953
954     // Add in non-leap years.
955     n = y
956     d += 365 * n
957
958     // Add in days before this month.
959     d += uint64(daysBefore[month-1])
960     if isLeap(year) && month >= March {
961         d++ // February 29
962     }
963
964     // Add in days before today.
965     d += uint64(day - 1)
966
967     // Add in time elapsed today.
968     abs := d * secondsPerDay
969     abs += uint64(hour*secondsPerHour + min*secondsPerMi
970
971     unix := int64(abs) + (absoluteToInternal + internalT
972
973     // Look for zone offset for t, so we can adjust to U
974     // The lookup function expects UTC, so we pass t in
975     // hope that it will not be too close to a zone tran
976     // and then adjust if it is.
977     _, offset, _, start, end := loc.lookup(unix)
978     if offset != 0 {
979         switch utc := unix - int64(offset); {
980             case utc < start:
981                 _, offset, _, _, _ = loc.lookup(star
982             case utc >= end:

```

```
983         _, offset, _, _, _ = loc.lookup(end)
984     }
985     unix -= int64(offset)
986 }
987
988     return Time{unix + unixToInternal, int32(nsec), loc}
989 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/time/zoneinfo.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package time
6
7 import (
8     "sync"
9     "syscall"
10 )
11
12 // A Location maps time instants to the zone in use at that
13 // Typically, the Location represents the collection of time
14 // in use in a geographical area, such as CEST and CET for c
15 type Location struct {
16     name string
17     zone []zone
18     tx   []zoneTrans
19
20     // Most lookups will be for the current time.
21     // To avoid the binary search through tx, keep a
22     // static one-element cache that gives the correct
23     // zone for the time when the Location was created.
24     // if cacheStart <= t <= cacheEnd,
25     // lookup can return cacheZone.
26     // The units for cacheStart and cacheEnd are seconds
27     // since January 1, 1970 UTC, to match the argument
28     // to lookup.
29     cacheStart int64
30     cacheEnd   int64
31     cacheZone  *zone
32 }
33
34 // A zone represents a single time zone such as CEST or CET.
35 type zone struct {
36     name    string // abbreviated name, "CET"
37     offset  int    // seconds east of UTC
38     isDST   bool   // is this zone Daylight Savings Time?
39 }
40
41 // A zoneTrans represents a single time zone transition.
42 type zoneTrans struct {
43     when      int64 // transition time, in seconds si
44     index     uint8 // the index of the zone that goe
```

```

45         isstd, isutc bool // ignored - no idea what these r
46     }
47
48 // UTC represents Universal Coordinated Time (UTC).
49 var UTC *Location = &utcLoc
50
51 // utcLoc is separate so that get can refer to &utcLoc
52 // and ensure that it never returns a nil *Location,
53 // even if a badly behaved client has changed UTC.
54 var utcLoc = Location{name: "UTC"}
55
56 // Local represents the system's local time zone.
57 var Local *Location = &localLoc
58
59 // localLoc is separate so that initLocal can initialize
60 // it even if a client has changed Local.
61 var localLoc Location
62 var localOnce sync.Once
63
64 func (l *Location) get() *Location {
65     if l == nil {
66         return &utcLoc
67     }
68     if l == &localLoc {
69         localOnce.Do(initLocal)
70     }
71     return l
72 }
73
74 // String returns a descriptive name for the time zone infor
75 // corresponding to the argument to LoadLocation.
76 func (l *Location) String() string {
77     return l.get().name
78 }
79
80 // FixedZone returns a Location that always uses
81 // the given zone name and offset (seconds east of UTC).
82 func FixedZone(name string, offset int) *Location {
83     l := &Location{
84         name:      name,
85         zone:      []zone{{name, offset, false}},
86         tx:        []zoneTrans{{-1 << 63, 0, false,
87         cacheStart: -1 << 63,
88         cacheEnd:  1<<63 - 1,
89     }
90     l.cacheZone = &l.zone[0]
91     return l
92 }
93
94 // lookup returns information about the time zone in use at

```

```

95 // instant in time expressed as seconds since January 1, 197
96 //
97 // The returned information gives the name of the zone (such
98 // the start and end times bracketing sec when that zone is
99 // the offset in seconds east of UTC (such as -5*60*60), and
100 // the daylight savings is being observed at that time.
101 func (l *Location) lookup(sec int64) (name string, offset in
102     l = l.get())
103
104     if len(l.tx) == 0 {
105         name = "UTC"
106         offset = 0
107         isDST = false
108         start = -1 << 63
109         end = 1<<63 - 1
110         return
111     }
112
113     if zone := l.cacheZone; zone != nil && l.cacheStart
114         name = zone.name
115         offset = zone.offset
116         isDST = zone.isDST
117         start = l.cacheStart
118         end = l.cacheEnd
119         return
120     }
121
122     // Binary search for entry with largest time <= sec.
123     // Not using sort.Search to avoid dependencies.
124     tx := l.tx
125     end = 1<<63 - 1
126     for len(tx) > 1 {
127         m := len(tx) / 2
128         lim := tx[m].when
129         if sec < lim {
130             end = lim
131             tx = tx[0:m]
132         } else {
133             tx = tx[m:]
134         }
135     }
136     zone := &l.zone[tx[0].index]
137     name = zone.name
138     offset = zone.offset
139     isDST = zone.isDST
140     start = tx[0].when
141     // end = maintained during the search
142     return
143 }

```

```

144
145 // lookupName returns information about the time zone with
146 // the given name (such as "EST").
147 func (l *Location) lookupName(name string) (offset int, isDS
148     l = l.get()
149     for i := range l.zone {
150         zone := &l.zone[i]
151         if zone.name == name {
152             return zone.offset, zone.isDST, true
153         }
154     }
155     return
156 }
157
158 // lookupOffset returns information about the time zone with
159 // the given offset (such as -5*60*60).
160 func (l *Location) lookupOffset(offset int) (name string, is
161     l = l.get()
162     for i := range l.zone {
163         zone := &l.zone[i]
164         if zone.offset == offset {
165             return zone.name, zone.isDST, true
166         }
167     }
168     return
169 }
170
171 // NOTE(rsc): Eventually we will need to accept the POSIX TZ
172 // syntax too, but I don't feel like implementing it today.
173
174 var zoneinfo, _ = syscall.Getenv("ZONEINFO")
175
176 // LoadLocation returns the Location with the given name.
177 //
178 // If the name is "" or "UTC", LoadLocation returns UTC.
179 // If the name is "Local", LoadLocation returns Local.
180 //
181 // Otherwise, the name is taken to be a location name corres
182 // in the IANA Time Zone database, such as "America/New_York
183 //
184 // The time zone database needed by LoadLocation may not be
185 // present on all systems, especially non-Unix systems.
186 // LoadLocation looks in the directory or uncompressed zip f
187 // named by the ZONEINFO environment variable, if any, then
188 // known installation locations on Unix systems,
189 // and finally looks in $GOROOT/lib/time/zoneinfo.zip.
190 func LoadLocation(name string) (*Location, error) {
191     if name == "" || name == "UTC" {
192         return UTC, nil

```

```
193     }
194     if name == "Local" {
195         return Local, nil
196     }
197     if zoneinfo != "" {
198         if z, err := loadZoneFile(zoneinfo, name); e
199             z.name = name
200             return z, nil
201         }
202     }
203     return loadLocation(name)
204 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/time/zoneinfo_read.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Parse "zoneinfo" time zone file.
6 // This is a fairly standard file format used on OS X, Linux
7 // See tzfile(5), http://en.wikipedia.org/wiki/Zoneinfo,
8 // and ftp://munnari.oz.au/pub/oldtz/
9
10 package time
11
12 import "errors"
13
14 const (
15     headerSize = 4 + 16 + 4*7
16 )
17
18 // Simple I/O interface to binary blob of data.
19 type data struct {
20     p []byte
21     error bool
22 }
23
24 func (d *data) read(n int) []byte {
25     if len(d.p) < n {
26         d.p = nil
27         d.error = true
28         return nil
29     }
30     p := d.p[0:n]
31     d.p = d.p[n:]
32     return p
33 }
34
35 func (d *data) big4() (n uint32, ok bool) {
36     p := d.read(4)
37     if len(p) < 4 {
38         d.error = true
39         return 0, false
40     }
41     return uint32(p[0])<<24 | uint32(p[1])<<16 | uint32(p[2])<<8 | uint32(p[3])
```

```

42 }
43
44 func (d *data) byte() (n byte, ok bool) {
45     p := d.read(1)
46     if len(p) < 1 {
47         d.error = true
48         return 0, false
49     }
50     return p[0], true
51 }
52
53 // Make a string by stopping at the first NUL
54 func byteString(p []byte) string {
55     for i := 0; i < len(p); i++ {
56         if p[i] == 0 {
57             return string(p[0:i])
58         }
59     }
60     return string(p)
61 }
62
63 var badData = errors.New("malformed time zone information")
64
65 func loadZoneData(bytes []byte) (l *Location, err error) {
66     d := data{bytes, false}
67
68     // 4-byte magic "TZif"
69     if magic := d.read(4); string(magic) != "TZif" {
70         return nil, badData
71     }
72
73     // 1-byte version, then 15 bytes of padding
74     var p []byte
75     if p = d.read(16); len(p) != 16 || p[0] != 0 && p[0]
76         return nil, badData
77     }
78
79     // six big-endian 32-bit integers:
80     //     number of UTC/local indicators
81     //     number of standard/wall indicators
82     //     number of leap seconds
83     //     number of transition times
84     //     number of local time zones
85     //     number of characters of time zone abbrev str
86     const (
87         NUTCLocal = iota
88         NStdWall
89         NLeap
90         NTime
91         NZone

```

```

92         NChar
93     )
94     var n [6]int
95     for i := 0; i < 6; i++ {
96         nn, ok := d.big4()
97         if !ok {
98             return nil, badData
99         }
100        n[i] = int(nn)
101    }
102
103    // Transition times.
104    txtimes := data{d.read(n[NTime] * 4), false}
105
106    // Time zone indices for transition times.
107    txzones := d.read(n[NTime])
108
109    // Zone info structures
110    zonedata := data{d.read(n[NZone] * 6), false}
111
112    // Time zone abbreviations.
113    abbrev := d.read(n[NChar])
114
115    // Leap-second time pairs
116    d.read(n[NLeap] * 8)
117
118    // Whether tx times associated with local time types
119    // are specified as standard time or wall time.
120    isstd := d.read(n[NStdWall])
121
122    // Whether tx times associated with local time types
123    // are specified as UTC or local time.
124    isutc := d.read(n[NUTCLocal])
125
126    if d.error { // ran out of data
127        return nil, badData
128    }
129
130    // If version == 2, the entire file repeats, this ti
131    // 8-byte ints for txtimes and leap seconds.
132    // We won't need those until 2106.
133
134    // Now we can build up a useful data structure.
135    // First the zone information.
136    //     utcoff[4] isdst[1] nameindex[1]
137    zone := make([]zone, n[NZone])
138    for i := range zone {
139        var ok bool
140        var n uint32

```

```

141         if n, ok = zonedata.big4(); !ok {
142             return nil, badData
143         }
144         zone[i].offset = int(n)
145         var b byte
146         if b, ok = zonedata.byte(); !ok {
147             return nil, badData
148         }
149         zone[i].isDST = b != 0
150         if b, ok = zonedata.byte(); !ok || int(b) >=
151             return nil, badData
152     }
153     zone[i].name = byteString(abbrev[b:])
154 }
155
156 // Now the transition time info.
157 tx := make([]zoneTrans, n[NTime])
158 for i := range tx {
159     var ok bool
160     var n uint32
161     if n, ok = txtimes.big4(); !ok {
162         return nil, badData
163     }
164     tx[i].when = int64(int32(n))
165     if int(txzones[i]) >= len(zone) {
166         return nil, badData
167     }
168     tx[i].index = txzones[i]
169     if i < len(isstd) {
170         tx[i].isstd = isstd[i] != 0
171     }
172     if i < len(isutc) {
173         tx[i].isutc = isutc[i] != 0
174     }
175 }
176
177 // Committed to succeed.
178 l = &Location{zone: zone, tx: tx}
179
180 // Fill in the cache with information about right no
181 // since that will be the most common lookup.
182 sec, _ := now()
183 for i := range tx {
184     if tx[i].when <= sec && (i+1 == len(tx) || s
185         l.cacheStart = tx[i].when
186         l.cacheEnd = 1<<63 - 1
187         if i+1 < len(tx) {
188             l.cacheEnd = tx[i+1].when
189         }

```

```

190             l.cacheZone = &l.zone[tx[i].index]
191         }
192     }
193
194     return l, nil
195 }
196
197 func loadZoneFile(dir, name string) (l *Location, err error) {
198     if len(dir) > 4 && dir[len(dir)-4:] == ".zip" {
199         return loadZoneZip(dir, name)
200     }
201     if dir != "" {
202         name = dir + "/" + name
203     }
204     buf, err := readFile(name)
205     if err != nil {
206         return
207     }
208     return loadZoneData(buf)
209 }
210
211 // There are 500+ zoneinfo files. Rather than distribute th
212 // individually, we ship them in an uncompressed zip file.
213 // Used this way, the zip file format serves as a commonly r
214 // container for the individual small files. We choose zip
215 // because zip files have a contiguous table of contents, ma
216 // individual file lookups faster, and because the per-file
217 // in a zip file is considerably less than tar's 512 bytes.
218
219 // get4 returns the little-endian 32-bit value in b.
220 func get4(b []byte) int {
221     if len(b) < 4 {
222         return 0
223     }
224     return int(b[0]) | int(b[1])<<8 | int(b[2])<<16 | in
225 }
226
227 // get2 returns the little-endian 16-bit value in b.
228 func get2(b []byte) int {
229     if len(b) < 2 {
230         return 0
231     }
232     return int(b[0]) | int(b[1])<<8
233 }
234
235 func loadZoneZip(zipfile, name string) (l *Location, err err
236     fd, err := open(zipfile)
237     if err != nil {
238         return nil, errors.New("open " + zipfile + "
239     }

```

```

240     defer closefd(fd)
241
242     const (
243         zeheader = 0x06054b50
244         zheader  = 0x02014b50
245         ztailsize = 22
246
247         zheadersize = 30
248         zheader      = 0x04034b50
249     )
250
251     buf := make([]byte, ztailsize)
252     if err := preadn(fd, buf, -ztailsize); err != nil ||
253         return nil, errors.New("corrupt zip file " +
254     }
255     n := get2(buf[10:])
256     size := get4(buf[12:])
257     off := get4(buf[16:])
258
259     buf = make([]byte, size)
260     if err := preadn(fd, buf, off); err != nil {
261         return nil, errors.New("corrupt zip file " +
262     }
263
264     for i := 0; i < n; i++ {
265         // zip entry layout:
266         //      0      magic[4]
267         //      4      madevers[1]
268         //      5      madeos[1]
269         //      6      extvers[1]
270         //      7      extos[1]
271         //      8      flags[2]
272         //     10      meth[2]
273         //     12      modtime[2]
274         //     14      moddate[2]
275         //     16      crc[4]
276         //     20      csize[4]
277         //     24      uncsiz[4]
278         //     28      namelen[2]
279         //     30      xlen[2]
280         //     32      fcrlen[2]
281         //     34      disknum[2]
282         //     36      iattr[2]
283         //     38      eattr[4]
284         //     42      off[4]
285         //     46      name[namelen]
286         //     46+namelen+xlen+fcrlen - next header
287         //
288         if get4(buf) != zheader {

```

```

289         break
290     }
291     meth := get2(buf[10:])
292     size := get4(buf[24:])
293     namelen := get2(buf[28:])
294     xlen := get2(buf[30:])
295     fclen := get2(buf[32:])
296     off := get4(buf[42:])
297     zname := buf[46 : 46+namelen]
298     buf = buf[46+namelen+xlen+fclen:]
299     if string(zname) != name {
300         continue
301     }
302     if meth != 0 {
303         return nil, errors.New("unsupported
304     }
305
306     // zip per-file header layout:
307     //      0      magic[4]
308     //      4      extvers[1]
309     //      5      extos[1]
310     //      6      flags[2]
311     //      8      meth[2]
312     //     10      modtime[2]
313     //     12      moddate[2]
314     //     14      crc[4]
315     //     18      csize[4]
316     //     22      uncsz[4]
317     //     26      namelen[2]
318     //     28      xlen[2]
319     //     30      name[namelen]
320     //     30+namelen+xlen - file data
321     //
322     buf = make([]byte, zheadersize+namelen)
323     if err := preadn(fd, buf, off); err != nil ||
324         get4(buf) != zheader ||
325         get2(buf[8:]) != meth ||
326         get2(buf[26:]) != namelen ||
327         string(buf[30:30+namelen]) != name {
328         return nil, errors.New("corrupt zip
329     }
330     xlen = get2(buf[28:])
331
332     buf = make([]byte, size)
333     if err := preadn(fd, buf, off+30+namelen+xle
334         return nil, errors.New("corrupt zip
335     }
336
337     return loadZoneData(buf)

```

```
338         }
339
340         return nil, errors.New("cannot find " + name + " in
341     }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/time/zoneinfo_unix.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // +build darwin freebsd linux netbsd openbsd
6
7 // Parse "zoneinfo" time zone file.
8 // This is a fairly standard file format used on OS X, Linux
9 // See tzfile(5), http://en.wikipedia.org/wiki/Zoneinfo,
10 // and ftp://munnari.oz.au/pub/oldtz/
11
12 package time
13
14 import (
15     "errors"
16     "runtime"
17     "syscall"
18 )
19
20 func initTestingZone() {
21     z, err := loadZoneFile(runtime.GOROOT()+"/lib/time/z
22     if err != nil {
23         panic("cannot load America/Los_Angeles for t
24     }
25     z.name = "Local"
26     localLoc = *z
27 }
28
29 // Many systems use /usr/share/zoneinfo, Solaris 2 has
30 // /usr/share/lib/zoneinfo, IRIX 6 has /usr/lib/locale/TZ.
31 var zoneDirs = []string{
32     "/usr/share/zoneinfo/",
33     "/usr/share/lib/zoneinfo/",
34     "/usr/lib/locale/TZ/",
35     runtime.GOROOT() + "/lib/time/zoneinfo/",
36 }
37
38 func initLocal() {
39     // consult $TZ to find the time zone to use.
40     // no $TZ means use the system default /etc/localtim
41     // $TZ="" means use UTC.
```

```

42     // $TZ="foo" means use /usr/share/zoneinfo/foo.
43
44     tz, ok := syscall.Getenv("TZ")
45     switch {
46     case !ok:
47         z, err := loadZoneFile("", "/etc/localtime")
48         if err == nil {
49             localLoc = *z
50             localLoc.name = "Local"
51             return
52         }
53     case tz != "" && tz != "UTC":
54         if z, err := loadLocation(tz); err == nil {
55             localLoc = *z
56             return
57         }
58     }
59
60     // Fall back to UTC.
61     localLoc.name = "UTC"
62 }
63
64 func loadLocation(name string) (*Location, error) {
65     for _, zoneDir := range zoneDirs {
66         if z, err := loadZoneFile(zoneDir, name); err == nil {
67             z.name = name
68             return z, nil
69         }
70     }
71     return nil, errors.New("unknown time zone " + name)
72 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/unicode/casetables.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // TODO: This file contains the special casing rules for Tur
6 // It should encompass all the languages with special casing
7 // and be generated automatically, but that requires some AP
8 // development first.
9
10 package unicode
11
12 var TurkishCase SpecialCase = _TurkishCase
13 var _TurkishCase = SpecialCase{
14     CaseRange{0x0049, 0x0049, d{0, 0x131 - 0x49, 0}},
15     CaseRange{0x0069, 0x0069, d{0x130 - 0x69, 0, 0x130 -
16     CaseRange{0x0130, 0x0130, d{0, 0x69 - 0x130, 0}},
17     CaseRange{0x0131, 0x0131, d{0x49 - 0x131, 0, 0x49 -
18 }
19
20 var AzeriCase SpecialCase = _TurkishCase
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/unicode/digit.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package unicode
6
7 // IsDigit reports whether the rune is a decimal digit.
8 func IsDigit(r rune) bool {
9     if r <= MaxLatin1 {
10         return '0' <= r && r <= '9'
11     }
12     return Is(Digit, r)
13 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/unicode/graphic.go

```
1 // Copyright 2011 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 package unicode
6
7 // Bit masks for each code point under U+0100, for fast look
8 const (
9     pC = 1 << iota // a control character.
10    pP              // a punctuation character.
11    pN              // a numeral.
12    pS              // a symbolic character.
13    pZ              // a spacing character.
14    pLu            // an upper-case letter.
15    pLl            // a lower-case letter.
16    pp            // a printable character according t
17    pg = pp | pZ // a graphical character according t
18 )
19
20 // GraphicRanges defines the set of graphic characters accor
21 var GraphicRanges = []*RangeTable{
22     L, M, N, P, S, Zs,
23 }
24
25 // PrintRanges defines the set of printable characters accor
26 // ASCII space, U+0020, is handled separately.
27 var PrintRanges = []*RangeTable{
28     L, M, N, P, S,
29 }
30
31 // IsGraphic reports whether the rune is defined as a Graphi
32 // Such characters include letters, marks, numbers, punctuat
33 // spaces, from categories L, M, N, P, S, Zs.
34 func IsGraphic(r rune) bool {
35     // We convert to uint32 to avoid the extra test for
36     // and in the index we convert to uint8 to avoid the
37     if uint32(r) <= MaxLatin1 {
38         return properties[uint8(r)]&pg != 0
39     }
40     return IsOneOf(GraphicRanges, r)
41 }
```

```

42
43 // IsPrint reports whether the rune is defined as printable
44 // characters include letters, marks, numbers, punctuation,
45 // ASCII space character, from categories L, M, N, P, S and
46 // character. This categorization is the same as IsGraphic
47 // only spacing character is ASCII space, U+0020.
48 func IsPrint(r rune) bool {
49     if uint32(r) <= MaxLatin1 {
50         return properties[uint8(r)]&pp != 0
51     }
52     return IsOneOf(PrintRanges, r)
53 }
54
55 // IsOneOf reports whether the rune is a member of one of th
56 func IsOneOf(set []*RangeTable, r rune) bool {
57     for _, inside := range set {
58         if Is(inside, r) {
59             return true
60         }
61     }
62     return false
63 }
64
65 // IsControl reports whether the rune is a control character
66 // The C (Other) Unicode category includes more code points
67 // such as surrogates; use Is(C, r) to test for them.
68 func IsControl(r rune) bool {
69     if uint32(r) <= MaxLatin1 {
70         return properties[uint8(r)]&pC != 0
71     }
72     // All control characters are < Latin1Max.
73     return false
74 }
75
76 // IsLetter reports whether the rune is a letter (category L
77 func IsLetter(r rune) bool {
78     if uint32(r) <= MaxLatin1 {
79         return properties[uint8(r)]&(pLu|pLl) != 0
80     }
81     return Is(Letter, r)
82 }
83
84 // IsMark reports whether the rune is a mark character (cate
85 func IsMark(r rune) bool {
86     // There are no mark characters in Latin-1.
87     return Is(Mark, r)
88 }
89
90 // IsNumber reports whether the rune is a number (category N
91 func IsNumber(r rune) bool {

```

```

92         if uint32(r) <= MaxLatin1 {
93             return properties[uint8(r)]&pN != 0
94         }
95         return Is(Number, r)
96     }
97
98     // IsPunct reports whether the rune is a Unicode punctuation
99     // (category P).
100    func IsPunct(r rune) bool {
101        if uint32(r) <= MaxLatin1 {
102            return properties[uint8(r)]&pP != 0
103        }
104        return Is(Punct, r)
105    }
106
107    // IsSpace reports whether the rune is a space character as
108    // by Unicode's White Space property; in the Latin-1 space
109    // this is
110    //     '\t', '\n', '\v', '\f', '\r', ' ', U+0085 (NEL), U+0
111    // Other definitions of spacing characters are set by catego
112    // Z and property Pattern_White_Space.
113    func IsSpace(r rune) bool {
114        // This property isn't the same as Z; special-case i
115        if uint32(r) <= MaxLatin1 {
116            switch r {
117                case '\t', '\n', '\v', '\f', '\r', ' ', 0x85
118            }
119            return true
120        }
121        return false
122    }
123    return Is(White_Space, r)
124 }
125
126 // IsSymbol reports whether the rune is a symbolic character
127 func IsSymbol(r rune) bool {
128     if uint32(r) <= MaxLatin1 {
129         return properties[uint8(r)]&pS != 0
130     }
131     return Is(Symbol, r)
132 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/unicode/letter.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package unicode provides data and functions to test some
6 // Unicode code points.
7 package unicode
8
9 const (
10     MaxRune          = '\U0010FFFF' // Maximum valid Unic
11     ReplacementChar = '\uFFFD'     // Represents invalid
12     MaxASCII        = '\u007F'     // maximum ASCII valu
13     MaxLatin1       = '\u00FF'     // maximum Latin-1 va
14 )
15
16 // RangeTable defines a set of Unicode code points by listin
17 // code points within the set. The ranges are listed in two
18 // to save space: a slice of 16-bit ranges and a slice of 32
19 // The two slices must be in sorted order and non-overlappin
20 // Also, R32 should contain only values >= 0x10000 (1<<16).
21 type RangeTable struct {
22     R16 []Range16
23     R32 []Range32
24 }
25
26 // Range16 represents of a range of 16-bit Unicode code poin
27 // inclusive and has the specified stride.
28 type Range16 struct {
29     Lo      uint16
30     Hi      uint16
31     Stride  uint16
32 }
33
34 // Range32 represents of a range of Unicode code points and
35 // more of the values will not fit in 16 bits. The range ru
36 // inclusive and has the specified stride. Lo and Hi must al
37 type Range32 struct {
38     Lo      uint32
39     Hi      uint32
40     Stride  uint32
41 }
42
43 // CaseRange represents a range of Unicode code points for s
44 // code point to one code point) case conversion.
```

```

45 // The range runs from Lo to Hi inclusive, with a fixed stri
46 // are the number to add to the code point to reach the code
47 // different case for that character. They may be negative.
48 // means the character is in the corresponding case. There i
49 // case representing sequences of alternating corresponding
50 // pairs. It appears with a fixed Delta of
51 //     {UpperLower, UpperLower, UpperLower}
52 // The constant UpperLower has an otherwise impossible delta
53 type CaseRange struct {
54     Lo    uint32
55     Hi    uint32
56     Delta d
57 }
58
59 // SpecialCase represents language-specific case mappings su
60 // Methods of SpecialCase customize (by overriding) the stan
61 type SpecialCase []CaseRange
62
63 // BUG(r): There is no mechanism for full case folding, that
64 // characters that involve multiple runes in the input or ou
65
66 // Indices into the Delta arrays inside CaseRanges for case
67 const (
68     UpperCase = iota
69     LowerCase
70     TitleCase
71     MaxCase
72 )
73
74 type d [MaxCase]rune // to make the CaseRanges text shorter
75
76 // If the Delta field of a CaseRange is UpperLower or LowerU
77 // this CaseRange represents a sequence of the form (say)
78 // Upper Lower Upper Lower.
79 const (
80     UpperLower = MaxRune + 1 // (Cannot be a valid delta
81 )
82
83 // is16 uses binary search to test whether rune is in the sp
84 func is16(ranges []Range16, r uint16) bool {
85     // binary search over ranges
86     lo := 0
87     hi := len(ranges)
88     for lo < hi {
89         m := lo + (hi-lo)/2
90         range_ := ranges[m]
91         if range_.Lo <= r && r <= range_.Hi {
92             return (r-range_.Lo)%range_.Stride =
93         }
94         if r < range_.Lo {

```

```

95             hi = m
96         } else {
97             lo = m + 1
98         }
99     }
100     return false
101 }
102
103 // is32 uses binary search to test whether rune is in the sp
104 func is32(ranges []Range32, r uint32) bool {
105     // binary search over ranges
106     lo := 0
107     hi := len(ranges)
108     for lo < hi {
109         m := lo + (hi-lo)/2
110         range_ := ranges[m]
111         if range_.Lo <= r && r <= range_.Hi {
112             return (r-range_.Lo)%range_.Stride =
113         }
114         if r < range_.Lo {
115             hi = m
116         } else {
117             lo = m + 1
118         }
119     }
120     return false
121 }
122
123 // Is tests whether rune is in the specified table of ranges
124 func Is(rangeTab *RangeTable, r rune) bool {
125     // common case: rune is ASCII or Latin-1.
126     if uint32(r) <= MaxLatin1 {
127         // Only need to check R16, since R32 is alwa
128         r16 := uint16(r)
129         for _, r := range rangeTab.R16 {
130             if r16 > r.Hi {
131                 continue
132             }
133             if r16 < r.Lo {
134                 return false
135             }
136             return (r16-r.Lo)%r.Stride == 0
137         }
138         return false
139     }
140     r16 := rangeTab.R16
141     if len(r16) > 0 && r <= rune(r16[len(r16)-1].Hi) {
142         return is16(r16, uint16(r))
143     }

```

```

144         r32 := rangeTab.R32
145         if len(r32) > 0 && r >= rune(r32[0].Lo) {
146             return is32(r32, uint32(r))
147         }
148         return false
149     }
150
151     // IsUpper reports whether the rune is an upper case letter.
152     func IsUpper(r rune) bool {
153         // See comment in IsGraphic.
154         if uint32(r) <= MaxLatin1 {
155             return properties[uint8(r)]&pLu != 0
156         }
157         return Is(Upper, r)
158     }
159
160     // IsLower reports whether the rune is a lower case letter.
161     func IsLower(r rune) bool {
162         // See comment in IsGraphic.
163         if uint32(r) <= MaxLatin1 {
164             return properties[uint8(r)]&pLl != 0
165         }
166         return Is(Lower, r)
167     }
168
169     // IsTitle reports whether the rune is a title case letter.
170     func IsTitle(r rune) bool {
171         if r <= MaxLatin1 {
172             return false
173         }
174         return Is(Title, r)
175     }
176
177     // to maps the rune using the specified case mapping.
178     func to(_case int, r rune, caseRange []CaseRange) rune {
179         if _case < 0 || MaxCase <= _case {
180             return ReplacementChar // as reasonable an e
181         }
182         // binary search over ranges
183         lo := 0
184         hi := len(caseRange)
185         for lo < hi {
186             m := lo + (hi-lo)/2
187             cr := caseRange[m]
188             if rune(cr.Lo) <= r && r <= rune(cr.Hi) {
189                 delta := rune(cr.Delta[_case])
190                 if delta > MaxRune {
191                     // In an Upper-Lower sequenc
192                     // an UpperCase letter, the

```

```

193             //      {0, 1, 0}   UpperCa
194             //      {-1, 0, -1} LowerCa
195             // The characters at even of
196             // sequence are upper case;
197             // The correct mapping can b
198             // bit in the sequence offse
199             // The constants UpperCase a
200             // is odd so we take the low
201             return rune(cr.Lo) + ((r-run
202             }
203             return r + delta
204         }
205         if r < rune(cr.Lo) {
206             hi = m
207         } else {
208             lo = m + 1
209         }
210     }
211     return r
212 }
213
214 // To maps the rune to the specified case: UpperCase, LowerC
215 func To(_case int, r rune) rune {
216     return to(_case, r, CaseRanges)
217 }
218
219 // ToUpper maps the rune to upper case.
220 func ToUpper(r rune) rune {
221     if r <= MaxASCII {
222         if 'a' <= r && r <= 'z' {
223             r -= 'a' - 'A'
224         }
225         return r
226     }
227     return To(UpperCase, r)
228 }
229
230 // ToLower maps the rune to lower case.
231 func ToLower(r rune) rune {
232     if r <= MaxASCII {
233         if 'A' <= r && r <= 'Z' {
234             r += 'a' - 'A'
235         }
236         return r
237     }
238     return To(LowerCase, r)
239 }
240
241 // ToTitle maps the rune to title case.
242 func ToTitle(r rune) rune {

```

```

243         if r <= MaxASCII {
244             if 'a' <= r && r <= 'z' { // title case is u
245                 r -= 'a' - 'A'
246             }
247             return r
248         }
249         return To(TitleCase, r)
250     }
251
252 // ToUpper maps the rune to upper case giving priority to th
253 func (special SpecialCase) ToUpper(r rune) rune {
254     r1 := to(UpperCase, r, []CaseRange(special))
255     if r1 == r {
256         r1 = ToUpper(r)
257     }
258     return r1
259 }
260
261 // ToTitle maps the rune to title case giving priority to th
262 func (special SpecialCase) ToTitle(r rune) rune {
263     r1 := to(TitleCase, r, []CaseRange(special))
264     if r1 == r {
265         r1 = ToTitle(r)
266     }
267     return r1
268 }
269
270 // ToLower maps the rune to lower case giving priority to th
271 func (special SpecialCase) ToLower(r rune) rune {
272     r1 := to(LowerCase, r, []CaseRange(special))
273     if r1 == r {
274         r1 = ToLower(r)
275     }
276     return r1
277 }
278
279 // caseOrbit is defined in tables.go as []foldPair. Right n
280 // entries fit in uint16, so use uint16. If that changes, c
281 // will fail (the constants in the composite literal will no
282 // and the types here can change to uint32.
283 type foldPair struct {
284     From uint16
285     To   uint16
286 }
287
288 // SimpleFold iterates over Unicode code points equivalent u
289 // the Unicode-defined simple case folding. Among the code
290 // equivalent to rune (including rune itself), SimpleFold re
291 // smallest rune >= r if one exists, or else the smallest ru

```

```

292 //
293 // For example:
294 //     SimpleFold('A') = 'a'
295 //     SimpleFold('a') = 'A'
296 //
297 //     SimpleFold('K') = 'k'
298 //     SimpleFold('k') = '\u212A' (Kelvin symbol, K)
299 //     SimpleFold('\u212A') = 'K'
300 //
301 //     SimpleFold('1') = '1'
302 //
303 func SimpleFold(r rune) rune {
304     // Consult caseOrbit table for special cases.
305     lo := 0
306     hi := len(caseOrbit)
307     for lo < hi {
308         m := lo + (hi-lo)/2
309         if rune(caseOrbit[m].From) < r {
310             lo = m + 1
311         } else {
312             hi = m
313         }
314     }
315     if lo < len(caseOrbit) && rune(caseOrbit[lo].From) =
316         return rune(caseOrbit[lo].To)
317     }
318
319     // No folding specified. This is a one- or two-elem
320     // equivalence class containing rune and ToLower(run
321     // and ToUpper(rune) if they are different from rune
322     if l := ToLower(r); l != r {
323         return l
324     }
325     return ToUpper(r)
326 }

```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).

[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/unicode/tables.go

```
1 // Generated by running
2 //     maketables --tables=all --data=http://www.unicode.or
3 // DO NOT EDIT
4
5 package unicode
6
7 // Version is the Unicode edition from which the tables are
8 const Version = "6.0.0"
9
10 // Categories is the set of Unicode category tables.
11 var Categories = map[string]*RangeTable{
12     "C": C,
13     "Cc": Cc,
14     "Cf": Cf,
15     "Co": Co,
16     "Cs": Cs,
17     "L": L,
18     "Ll": Ll,
19     "Lm": Lm,
20     "Lo": Lo,
21     "Lt": Lt,
22     "Lu": Lu,
23     "M": M,
24     "Mc": Mc,
25     "Me": Me,
26     "Mn": Mn,
27     "N": N,
28     "Nd": Nd,
29     "Nl": Nl,
30     "No": No,
31     "P": P,
32     "Pc": Pc,
33     "Pd": Pd,
34     "Pe": Pe,
35     "Pf": Pf,
36     "Pi": Pi,
37     "Po": Po,
38     "Ps": Ps,
39     "S": S,
40     "Sc": Sc,
41     "Sk": Sk,
42     "Sm": Sm,
43     "So": So,
44     "Z": Z,
```

```

45         "Zl": Zl,
46         "Zp": Zp,
47         "Zs": Zs,
48     }
49
50     var _C = &RangeTable{
51         R16: []Range16{
52             {0x0001, 0x001f, 1},
53             {0x007f, 0x009f, 1},
54             {0x00ad, 0x0600, 1363},
55             {0x0601, 0x0603, 1},
56             {0x06dd, 0x070f, 50},
57             {0x17b4, 0x17b5, 1},
58             {0x200b, 0x200f, 1},
59             {0x202a, 0x202e, 1},
60             {0x2060, 0x2064, 1},
61             {0x206a, 0x206f, 1},
62             {0xd800, 0xf8ff, 1},
63             {0xfeff, 0xffff9, 250},
64             {0xffffa, 0xffffb, 1},
65         },
66         R32: []Range32{
67             {0x110bd, 0x1d173, 49334},
68             {0x1d174, 0x1d17a, 1},
69             {0xe0001, 0xe0020, 31},
70             {0xe0021, 0xe007f, 1},
71             {0xf0000, 0xfffffd, 1},
72             {0x100000, 0x10ffffd, 1},
73         },
74     }
75
76     var _Cc = &RangeTable{
77         R16: []Range16{
78             {0x0001, 0x001f, 1},
79             {0x007f, 0x009f, 1},
80         },
81     }
82
83     var _Cf = &RangeTable{
84         R16: []Range16{
85             {0x00ad, 0x0600, 1363},
86             {0x0601, 0x0603, 1},
87             {0x06dd, 0x070f, 50},
88             {0x17b4, 0x17b5, 1},
89             {0x200b, 0x200f, 1},
90             {0x202a, 0x202e, 1},
91             {0x2060, 0x2064, 1},
92             {0x206a, 0x206f, 1},
93             {0xfeff, 0xffff9, 250},
94             {0xffffa, 0xffffb, 1},

```

```

95     },
96     R32: []Range32{
97         {0x110bd, 0x1d173, 49334},
98         {0x1d174, 0x1d17a, 1},
99         {0xe0001, 0xe0020, 31},
100        {0xe0021, 0xe007f, 1},
101    },
102 }
103
104 var _Co = &RangeTable{
105     R16: []Range16{
106         {0xe000, 0xf8ff, 1},
107     },
108     R32: []Range32{
109         {0xf0000, 0xffffd, 1},
110         {0x100000, 0x10ffffd, 1},
111     },
112 }
113
114 var _Cs = &RangeTable{
115     R16: []Range16{
116         {0xd800, 0xdfff, 1},
117     },
118 }
119
120 var _L = &RangeTable{
121     R16: []Range16{
122         {0x0041, 0x005a, 1},
123         {0x0061, 0x007a, 1},
124         {0x00aa, 0x00b5, 11},
125         {0x00ba, 0x00c0, 6},
126         {0x00c1, 0x00d6, 1},
127         {0x00d8, 0x00f6, 1},
128         {0x00f8, 0x02c1, 1},
129         {0x02c6, 0x02d1, 1},
130         {0x02e0, 0x02e4, 1},
131         {0x02ec, 0x02ee, 2},
132         {0x0370, 0x0374, 1},
133         {0x0376, 0x0377, 1},
134         {0x037a, 0x037d, 1},
135         {0x0386, 0x0388, 2},
136         {0x0389, 0x038a, 1},
137         {0x038c, 0x038e, 2},
138         {0x038f, 0x03a1, 1},
139         {0x03a3, 0x03f5, 1},
140         {0x03f7, 0x0481, 1},
141         {0x048a, 0x0527, 1},
142         {0x0531, 0x0556, 1},
143         {0x0559, 0x0561, 8},

```

144	{0x0562, 0x0587, 1},
145	{0x05d0, 0x05ea, 1},
146	{0x05f0, 0x05f2, 1},
147	{0x0620, 0x064a, 1},
148	{0x066e, 0x066f, 1},
149	{0x0671, 0x06d3, 1},
150	{0x06d5, 0x06e5, 16},
151	{0x06e6, 0x06ee, 8},
152	{0x06ef, 0x06fa, 11},
153	{0x06fb, 0x06fc, 1},
154	{0x06ff, 0x0710, 17},
155	{0x0712, 0x072f, 1},
156	{0x074d, 0x07a5, 1},
157	{0x07b1, 0x07ca, 25},
158	{0x07cb, 0x07ea, 1},
159	{0x07f4, 0x07f5, 1},
160	{0x07fa, 0x0800, 6},
161	{0x0801, 0x0815, 1},
162	{0x081a, 0x0824, 10},
163	{0x0828, 0x0840, 24},
164	{0x0841, 0x0858, 1},
165	{0x0904, 0x0939, 1},
166	{0x093d, 0x0950, 19},
167	{0x0958, 0x0961, 1},
168	{0x0971, 0x0977, 1},
169	{0x0979, 0x097f, 1},
170	{0x0985, 0x098c, 1},
171	{0x098f, 0x0990, 1},
172	{0x0993, 0x09a8, 1},
173	{0x09aa, 0x09b0, 1},
174	{0x09b2, 0x09b6, 4},
175	{0x09b7, 0x09b9, 1},
176	{0x09bd, 0x09ce, 17},
177	{0x09dc, 0x09dd, 1},
178	{0x09df, 0x09e1, 1},
179	{0x09f0, 0x09f1, 1},
180	{0x0a05, 0x0a0a, 1},
181	{0x0a0f, 0x0a10, 1},
182	{0x0a13, 0x0a28, 1},
183	{0x0a2a, 0x0a30, 1},
184	{0x0a32, 0x0a33, 1},
185	{0x0a35, 0x0a36, 1},
186	{0x0a38, 0x0a39, 1},
187	{0x0a59, 0x0a5c, 1},
188	{0x0a5e, 0x0a72, 20},
189	{0x0a73, 0x0a74, 1},
190	{0x0a85, 0x0a8d, 1},
191	{0x0a8f, 0x0a91, 1},
192	{0x0a93, 0x0aa8, 1},

193 {0x0aaa, 0x0ab0, 1},
194 {0x0ab2, 0x0ab3, 1},
195 {0x0ab5, 0x0ab9, 1},
196 {0x0abd, 0x0ad0, 19},
197 {0x0ae0, 0x0ae1, 1},
198 {0x0b05, 0x0b0c, 1},
199 {0x0b0f, 0x0b10, 1},
200 {0x0b13, 0x0b28, 1},
201 {0x0b2a, 0x0b30, 1},
202 {0x0b32, 0x0b33, 1},
203 {0x0b35, 0x0b39, 1},
204 {0x0b3d, 0x0b5c, 31},
205 {0x0b5d, 0x0b5f, 2},
206 {0x0b60, 0x0b61, 1},
207 {0x0b71, 0x0b83, 18},
208 {0x0b85, 0x0b8a, 1},
209 {0x0b8e, 0x0b90, 1},
210 {0x0b92, 0x0b95, 1},
211 {0x0b99, 0x0b9a, 1},
212 {0x0b9c, 0x0b9e, 2},
213 {0x0b9f, 0x0ba3, 4},
214 {0x0ba4, 0x0ba8, 4},
215 {0x0ba9, 0x0baa, 1},
216 {0x0bae, 0x0bb9, 1},
217 {0x0bd0, 0x0c05, 53},
218 {0x0c06, 0x0c0c, 1},
219 {0x0c0e, 0x0c10, 1},
220 {0x0c12, 0x0c28, 1},
221 {0x0c2a, 0x0c33, 1},
222 {0x0c35, 0x0c39, 1},
223 {0x0c3d, 0x0c58, 27},
224 {0x0c59, 0x0c60, 7},
225 {0x0c61, 0x0c85, 36},
226 {0x0c86, 0x0c8c, 1},
227 {0x0c8e, 0x0c90, 1},
228 {0x0c92, 0x0ca8, 1},
229 {0x0caa, 0x0cb3, 1},
230 {0x0cb5, 0x0cb9, 1},
231 {0x0cbd, 0x0cde, 33},
232 {0x0ce0, 0x0ce1, 1},
233 {0x0cf1, 0x0cf2, 1},
234 {0x0d05, 0x0d0c, 1},
235 {0x0d0e, 0x0d10, 1},
236 {0x0d12, 0x0d3a, 1},
237 {0x0d3d, 0x0d4e, 17},
238 {0x0d60, 0x0d61, 1},
239 {0x0d7a, 0x0d7f, 1},
240 {0x0d85, 0x0d96, 1},
241 {0x0d9a, 0x0db1, 1},
242 {0x0db3, 0x0dbb, 1},

243	{0x0dbd, 0x0dc0, 3},
244	{0x0dc1, 0x0dc6, 1},
245	{0x0e01, 0x0e30, 1},
246	{0x0e32, 0x0e33, 1},
247	{0x0e40, 0x0e46, 1},
248	{0x0e81, 0x0e82, 1},
249	{0x0e84, 0x0e87, 3},
250	{0x0e88, 0x0e8a, 2},
251	{0x0e8d, 0x0e94, 7},
252	{0x0e95, 0x0e97, 1},
253	{0x0e99, 0x0e9f, 1},
254	{0x0ea1, 0x0ea3, 1},
255	{0x0ea5, 0x0ea7, 2},
256	{0x0eaa, 0x0eab, 1},
257	{0x0ead, 0x0eb0, 1},
258	{0x0eb2, 0x0eb3, 1},
259	{0x0ebd, 0x0ec0, 3},
260	{0x0ec1, 0x0ec4, 1},
261	{0x0ec6, 0x0edc, 22},
262	{0x0edd, 0x0f00, 35},
263	{0x0f40, 0x0f47, 1},
264	{0x0f49, 0x0f6c, 1},
265	{0x0f88, 0x0f8c, 1},
266	{0x1000, 0x102a, 1},
267	{0x103f, 0x1050, 17},
268	{0x1051, 0x1055, 1},
269	{0x105a, 0x105d, 1},
270	{0x1061, 0x1065, 4},
271	{0x1066, 0x106e, 8},
272	{0x106f, 0x1070, 1},
273	{0x1075, 0x1081, 1},
274	{0x108e, 0x10a0, 18},
275	{0x10a1, 0x10c5, 1},
276	{0x10d0, 0x10fa, 1},
277	{0x10fc, 0x1100, 4},
278	{0x1101, 0x1248, 1},
279	{0x124a, 0x124d, 1},
280	{0x1250, 0x1256, 1},
281	{0x1258, 0x125a, 2},
282	{0x125b, 0x125d, 1},
283	{0x1260, 0x1288, 1},
284	{0x128a, 0x128d, 1},
285	{0x1290, 0x12b0, 1},
286	{0x12b2, 0x12b5, 1},
287	{0x12b8, 0x12be, 1},
288	{0x12c0, 0x12c2, 2},
289	{0x12c3, 0x12c5, 1},
290	{0x12c8, 0x12d6, 1},
291	{0x12d8, 0x1310, 1},

292 {0x1312, 0x1315, 1},
293 {0x1318, 0x135a, 1},
294 {0x1380, 0x138f, 1},
295 {0x13a0, 0x13f4, 1},
296 {0x1401, 0x166c, 1},
297 {0x166f, 0x167f, 1},
298 {0x1681, 0x169a, 1},
299 {0x16a0, 0x16ea, 1},
300 {0x1700, 0x170c, 1},
301 {0x170e, 0x1711, 1},
302 {0x1720, 0x1731, 1},
303 {0x1740, 0x1751, 1},
304 {0x1760, 0x176c, 1},
305 {0x176e, 0x1770, 1},
306 {0x1780, 0x17b3, 1},
307 {0x17d7, 0x17dc, 5},
308 {0x1820, 0x1877, 1},
309 {0x1880, 0x18a8, 1},
310 {0x18aa, 0x18b0, 6},
311 {0x18b1, 0x18f5, 1},
312 {0x1900, 0x191c, 1},
313 {0x1950, 0x196d, 1},
314 {0x1970, 0x1974, 1},
315 {0x1980, 0x19ab, 1},
316 {0x19c1, 0x19c7, 1},
317 {0x1a00, 0x1a16, 1},
318 {0x1a20, 0x1a54, 1},
319 {0x1aa7, 0x1b05, 94},
320 {0x1b06, 0x1b33, 1},
321 {0x1b45, 0x1b4b, 1},
322 {0x1b83, 0x1ba0, 1},
323 {0x1bae, 0x1baf, 1},
324 {0x1bc0, 0x1be5, 1},
325 {0x1c00, 0x1c23, 1},
326 {0x1c4d, 0x1c4f, 1},
327 {0x1c5a, 0x1c7d, 1},
328 {0x1ce9, 0x1cec, 1},
329 {0x1cee, 0x1cf1, 1},
330 {0x1d00, 0x1dbf, 1},
331 {0x1e00, 0x1f15, 1},
332 {0x1f18, 0x1f1d, 1},
333 {0x1f20, 0x1f45, 1},
334 {0x1f48, 0x1f4d, 1},
335 {0x1f50, 0x1f57, 1},
336 {0x1f59, 0x1f5f, 2},
337 {0x1f60, 0x1f7d, 1},
338 {0x1f80, 0x1fb4, 1},
339 {0x1fb6, 0x1fbc, 1},
340 {0x1fbe, 0x1fc2, 4},

341 {0x1fc3, 0x1fc4, 1},
342 {0x1fc6, 0x1fcc, 1},
343 {0x1fd0, 0x1fd3, 1},
344 {0x1fd6, 0x1fdb, 1},
345 {0x1fe0, 0x1fec, 1},
346 {0x1ff2, 0x1ff4, 1},
347 {0x1ff6, 0x1ffc, 1},
348 {0x2071, 0x207f, 14},
349 {0x2090, 0x209c, 1},
350 {0x2102, 0x2107, 5},
351 {0x210a, 0x2113, 1},
352 {0x2115, 0x2119, 4},
353 {0x211a, 0x211d, 1},
354 {0x2124, 0x212a, 2},
355 {0x212b, 0x212d, 1},
356 {0x212f, 0x2139, 1},
357 {0x213c, 0x213f, 1},
358 {0x2145, 0x2149, 1},
359 {0x214e, 0x2183, 53},
360 {0x2184, 0x2c00, 2684},
361 {0x2c01, 0x2c2e, 1},
362 {0x2c30, 0x2c5e, 1},
363 {0x2c60, 0x2ce4, 1},
364 {0x2ceb, 0x2cee, 1},
365 {0x2d00, 0x2d25, 1},
366 {0x2d30, 0x2d65, 1},
367 {0x2d6f, 0x2d80, 17},
368 {0x2d81, 0x2d96, 1},
369 {0x2da0, 0x2da6, 1},
370 {0x2da8, 0x2dae, 1},
371 {0x2db0, 0x2db6, 1},
372 {0x2db8, 0x2dbe, 1},
373 {0x2dc0, 0x2dc6, 1},
374 {0x2dc8, 0x2dce, 1},
375 {0x2dd0, 0x2dd6, 1},
376 {0x2dd8, 0x2dde, 1},
377 {0x2e2f, 0x3005, 470},
378 {0x3006, 0x3031, 43},
379 {0x3032, 0x3035, 1},
380 {0x303b, 0x303c, 1},
381 {0x3041, 0x3096, 1},
382 {0x309d, 0x309f, 1},
383 {0x30a1, 0x30fa, 1},
384 {0x30fc, 0x30ff, 1},
385 {0x3105, 0x312d, 1},
386 {0x3131, 0x318e, 1},
387 {0x31a0, 0x31ba, 1},
388 {0x31f0, 0x31ff, 1},
389 {0x3400, 0x4db5, 1},
390 {0x4e00, 0x9fcb, 1},

391 {0xa000, 0xa48c, 1},
392 {0xa4d0, 0xa4fd, 1},
393 {0xa500, 0xa60c, 1},
394 {0xa610, 0xa61f, 1},
395 {0xa62a, 0xa62b, 1},
396 {0xa640, 0xa66e, 1},
397 {0xa67f, 0xa697, 1},
398 {0xa6a0, 0xa6e5, 1},
399 {0xa717, 0xa71f, 1},
400 {0xa722, 0xa788, 1},
401 {0xa78b, 0xa78e, 1},
402 {0xa790, 0xa791, 1},
403 {0xa7a0, 0xa7a9, 1},
404 {0xa7fa, 0xa801, 1},
405 {0xa803, 0xa805, 1},
406 {0xa807, 0xa80a, 1},
407 {0xa80c, 0xa822, 1},
408 {0xa840, 0xa873, 1},
409 {0xa882, 0xa8b3, 1},
410 {0xa8f2, 0xa8f7, 1},
411 {0xa8fb, 0xa90a, 15},
412 {0xa90b, 0xa925, 1},
413 {0xa930, 0xa946, 1},
414 {0xa960, 0xa97c, 1},
415 {0xa984, 0xa9b2, 1},
416 {0xa9cf, 0xaa00, 49},
417 {0xaa01, 0xaa28, 1},
418 {0xaa40, 0xaa42, 1},
419 {0xaa44, 0xaa4b, 1},
420 {0xaa60, 0xaa76, 1},
421 {0xaa7a, 0xaa80, 6},
422 {0xaa81, 0xaaaf, 1},
423 {0xaab1, 0xaab5, 4},
424 {0xaab6, 0xaab9, 3},
425 {0xaaba, 0xaabd, 1},
426 {0xaac0, 0xaac2, 2},
427 {0xaadb, 0xaadd, 1},
428 {0xab01, 0xab06, 1},
429 {0xab09, 0xab0e, 1},
430 {0xab11, 0xab16, 1},
431 {0xab20, 0xab26, 1},
432 {0xab28, 0xab2e, 1},
433 {0xabc0, 0xabe2, 1},
434 {0xac00, 0xd7a3, 1},
435 {0xd7b0, 0xd7c6, 1},
436 {0xd7cb, 0xd7fb, 1},
437 {0xf900, 0xfa2d, 1},
438 {0xfa30, 0xfa6d, 1},
439 {0xfa70, 0xfad9, 1},

```

440      {0xfb00, 0xfb06, 1},
441      {0xfb13, 0xfb17, 1},
442      {0xfb1d, 0xfb1f, 2},
443      {0xfb20, 0xfb28, 1},
444      {0xfb2a, 0xfb36, 1},
445      {0xfb38, 0xfb3c, 1},
446      {0xfb3e, 0xfb40, 2},
447      {0xfb41, 0xfb43, 2},
448      {0xfb44, 0xfb46, 2},
449      {0xfb47, 0xfbb1, 1},
450      {0xfbd3, 0xfd3d, 1},
451      {0xfd50, 0xfd8f, 1},
452      {0xfd92, 0xfdc7, 1},
453      {0xfdf0, 0xfdfb, 1},
454      {0xfe70, 0xfe74, 1},
455      {0xfe76, 0xfefc, 1},
456      {0xff21, 0xff3a, 1},
457      {0xff41, 0xff5a, 1},
458      {0xff66, 0xffbe, 1},
459      {0xffc2, 0xffc7, 1},
460      {0xffca, 0xffcf, 1},
461      {0xffd2, 0xffd7, 1},
462      {0xffda, 0xffdc, 1},
463      },
464      R32: []Range32{
465          {0x10000, 0x1000b, 1},
466          {0x1000d, 0x10026, 1},
467          {0x10028, 0x1003a, 1},
468          {0x1003c, 0x1003d, 1},
469          {0x1003f, 0x1004d, 1},
470          {0x10050, 0x1005d, 1},
471          {0x10080, 0x100fa, 1},
472          {0x10280, 0x1029c, 1},
473          {0x102a0, 0x102d0, 1},
474          {0x10300, 0x1031e, 1},
475          {0x10330, 0x10340, 1},
476          {0x10342, 0x10349, 1},
477          {0x10380, 0x1039d, 1},
478          {0x103a0, 0x103c3, 1},
479          {0x103c8, 0x103cf, 1},
480          {0x10400, 0x1049d, 1},
481          {0x10800, 0x10805, 1},
482          {0x10808, 0x1080a, 2},
483          {0x1080b, 0x10835, 1},
484          {0x10837, 0x10838, 1},
485          {0x1083c, 0x1083f, 3},
486          {0x10840, 0x10855, 1},
487          {0x10900, 0x10915, 1},
488          {0x10920, 0x10939, 1},

```

```
489      {0x10a00, 0x10a10, 16},
490      {0x10a11, 0x10a13, 1},
491      {0x10a15, 0x10a17, 1},
492      {0x10a19, 0x10a33, 1},
493      {0x10a60, 0x10a7c, 1},
494      {0x10b00, 0x10b35, 1},
495      {0x10b40, 0x10b55, 1},
496      {0x10b60, 0x10b72, 1},
497      {0x10c00, 0x10c48, 1},
498      {0x11003, 0x11037, 1},
499      {0x11083, 0x110af, 1},
500      {0x12000, 0x1236e, 1},
501      {0x13000, 0x1342e, 1},
502      {0x16800, 0x16a38, 1},
503      {0x1b000, 0x1b001, 1},
504      {0x1d400, 0x1d454, 1},
505      {0x1d456, 0x1d49c, 1},
506      {0x1d49e, 0x1d49f, 1},
507      {0x1d4a2, 0x1d4a5, 3},
508      {0x1d4a6, 0x1d4a9, 3},
509      {0x1d4aa, 0x1d4ac, 1},
510      {0x1d4ae, 0x1d4b9, 1},
511      {0x1d4bb, 0x1d4bd, 2},
512      {0x1d4be, 0x1d4c3, 1},
513      {0x1d4c5, 0x1d505, 1},
514      {0x1d507, 0x1d50a, 1},
515      {0x1d50d, 0x1d514, 1},
516      {0x1d516, 0x1d51c, 1},
517      {0x1d51e, 0x1d539, 1},
518      {0x1d53b, 0x1d53e, 1},
519      {0x1d540, 0x1d544, 1},
520      {0x1d546, 0x1d54a, 4},
521      {0x1d54b, 0x1d550, 1},
522      {0x1d552, 0x1d6a5, 1},
523      {0x1d6a8, 0x1d6c0, 1},
524      {0x1d6c2, 0x1d6da, 1},
525      {0x1d6dc, 0x1d6fa, 1},
526      {0x1d6fc, 0x1d714, 1},
527      {0x1d716, 0x1d734, 1},
528      {0x1d736, 0x1d74e, 1},
529      {0x1d750, 0x1d76e, 1},
530      {0x1d770, 0x1d788, 1},
531      {0x1d78a, 0x1d7a8, 1},
532      {0x1d7aa, 0x1d7c2, 1},
533      {0x1d7c4, 0x1d7cb, 1},
534      {0x20000, 0x2a6d6, 1},
535      {0x2a700, 0x2b734, 1},
536      {0x2b740, 0x2b81d, 1},
537      {0x2f800, 0x2fa1d, 1},
538      },
```

```
539 }
540
541 var _L1 = &RangeTable{
542     R16: []Range16{
543         {0x0061, 0x007a, 1},
544         {0x00aa, 0x00b5, 11},
545         {0x00ba, 0x00df, 37},
546         {0x00e0, 0x00f6, 1},
547         {0x00f8, 0x00ff, 1},
548         {0x0101, 0x0137, 2},
549         {0x0138, 0x0148, 2},
550         {0x0149, 0x0177, 2},
551         {0x017a, 0x017e, 2},
552         {0x017f, 0x0180, 1},
553         {0x0183, 0x0185, 2},
554         {0x0188, 0x018c, 4},
555         {0x018d, 0x0192, 5},
556         {0x0195, 0x0199, 4},
557         {0x019a, 0x019b, 1},
558         {0x019e, 0x01a1, 3},
559         {0x01a3, 0x01a5, 2},
560         {0x01a8, 0x01aa, 2},
561         {0x01ab, 0x01ad, 2},
562         {0x01b0, 0x01b4, 4},
563         {0x01b6, 0x01b9, 3},
564         {0x01ba, 0x01bd, 3},
565         {0x01be, 0x01bf, 1},
566         {0x01c6, 0x01cc, 3},
567         {0x01ce, 0x01dc, 2},
568         {0x01dd, 0x01ef, 2},
569         {0x01f0, 0x01f3, 3},
570         {0x01f5, 0x01f9, 4},
571         {0x01fb, 0x0233, 2},
572         {0x0234, 0x0239, 1},
573         {0x023c, 0x023f, 3},
574         {0x0240, 0x0242, 2},
575         {0x0247, 0x024f, 2},
576         {0x0250, 0x0293, 1},
577         {0x0295, 0x02af, 1},
578         {0x0371, 0x0373, 2},
579         {0x0377, 0x037b, 4},
580         {0x037c, 0x037d, 1},
581         {0x0390, 0x03ac, 28},
582         {0x03ad, 0x03ce, 1},
583         {0x03d0, 0x03d1, 1},
584         {0x03d5, 0x03d7, 1},
585         {0x03d9, 0x03ef, 2},
586         {0x03f0, 0x03f3, 1},
587         {0x03f5, 0x03fb, 3},
```

588 {0x03fc, 0x0430, 52},
589 {0x0431, 0x045f, 1},
590 {0x0461, 0x0481, 2},
591 {0x048b, 0x04bf, 2},
592 {0x04c2, 0x04ce, 2},
593 {0x04cf, 0x0527, 2},
594 {0x0561, 0x0587, 1},
595 {0x1d00, 0x1d2b, 1},
596 {0x1d62, 0x1d77, 1},
597 {0x1d79, 0x1d9a, 1},
598 {0x1e01, 0x1e95, 2},
599 {0x1e96, 0x1e9d, 1},
600 {0x1e9f, 0x1eff, 2},
601 {0x1f00, 0x1f07, 1},
602 {0x1f10, 0x1f15, 1},
603 {0x1f20, 0x1f27, 1},
604 {0x1f30, 0x1f37, 1},
605 {0x1f40, 0x1f45, 1},
606 {0x1f50, 0x1f57, 1},
607 {0x1f60, 0x1f67, 1},
608 {0x1f70, 0x1f7d, 1},
609 {0x1f80, 0x1f87, 1},
610 {0x1f90, 0x1f97, 1},
611 {0x1fa0, 0x1fa7, 1},
612 {0x1fb0, 0x1fb4, 1},
613 {0x1fb6, 0x1fb7, 1},
614 {0x1fbe, 0x1fc2, 4},
615 {0x1fc3, 0x1fc4, 1},
616 {0x1fc6, 0x1fc7, 1},
617 {0x1fd0, 0x1fd3, 1},
618 {0x1fd6, 0x1fd7, 1},
619 {0x1fe0, 0x1fe7, 1},
620 {0x1ff2, 0x1ff4, 1},
621 {0x1ff6, 0x1ff7, 1},
622 {0x210a, 0x210e, 4},
623 {0x210f, 0x2113, 4},
624 {0x212f, 0x2139, 5},
625 {0x213c, 0x213d, 1},
626 {0x2146, 0x2149, 1},
627 {0x214e, 0x2184, 54},
628 {0x2c30, 0x2c5e, 1},
629 {0x2c61, 0x2c65, 4},
630 {0x2c66, 0x2c6c, 2},
631 {0x2c71, 0x2c73, 2},
632 {0x2c74, 0x2c76, 2},
633 {0x2c77, 0x2c7c, 1},
634 {0x2c81, 0x2ce3, 2},
635 {0x2ce4, 0x2cec, 8},
636 {0x2cee, 0x2d00, 18},

```

637         {0x2d01, 0x2d25, 1},
638         {0xa641, 0xa66d, 2},
639         {0xa681, 0xa697, 2},
640         {0xa723, 0xa72f, 2},
641         {0xa730, 0xa731, 1},
642         {0xa733, 0xa771, 2},
643         {0xa772, 0xa778, 1},
644         {0xa77a, 0xa77c, 2},
645         {0xa77f, 0xa787, 2},
646         {0xa78c, 0xa78e, 2},
647         {0xa791, 0xa7a1, 16},
648         {0xa7a3, 0xa7a9, 2},
649         {0xa7fa, 0xfb00, 21254},
650         {0xfb01, 0xfb06, 1},
651         {0xfb13, 0xfb17, 1},
652         {0xff41, 0xff5a, 1},
653     },
654     R32: []Range32{
655         {0x10428, 0x1044f, 1},
656         {0x1d41a, 0x1d433, 1},
657         {0x1d44e, 0x1d454, 1},
658         {0x1d456, 0x1d467, 1},
659         {0x1d482, 0x1d49b, 1},
660         {0x1d4b6, 0x1d4b9, 1},
661         {0x1d4bb, 0x1d4bd, 2},
662         {0x1d4be, 0x1d4c3, 1},
663         {0x1d4c5, 0x1d4cf, 1},
664         {0x1d4ea, 0x1d503, 1},
665         {0x1d51e, 0x1d537, 1},
666         {0x1d552, 0x1d56b, 1},
667         {0x1d586, 0x1d59f, 1},
668         {0x1d5ba, 0x1d5d3, 1},
669         {0x1d5ee, 0x1d607, 1},
670         {0x1d622, 0x1d63b, 1},
671         {0x1d656, 0x1d66f, 1},
672         {0x1d68a, 0x1d6a5, 1},
673         {0x1d6c2, 0x1d6da, 1},
674         {0x1d6dc, 0x1d6e1, 1},
675         {0x1d6fc, 0x1d714, 1},
676         {0x1d716, 0x1d71b, 1},
677         {0x1d736, 0x1d74e, 1},
678         {0x1d750, 0x1d755, 1},
679         {0x1d770, 0x1d788, 1},
680         {0x1d78a, 0x1d78f, 1},
681         {0x1d7aa, 0x1d7c2, 1},
682         {0x1d7c4, 0x1d7c9, 1},
683         {0x1d7cb, 0x1d7cb, 1},
684     },
685 }
686

```

```

687 var _Lm = &RangeTable{
688     R16: []Range16{
689         {0x02b0, 0x02c1, 1},
690         {0x02c6, 0x02d1, 1},
691         {0x02e0, 0x02e4, 1},
692         {0x02ec, 0x02ee, 2},
693         {0x0374, 0x037a, 6},
694         {0x0559, 0x0640, 231},
695         {0x06e5, 0x06e6, 1},
696         {0x07f4, 0x07f5, 1},
697         {0x07fa, 0x081a, 32},
698         {0x0824, 0x0828, 4},
699         {0x0971, 0x0e46, 1237},
700         {0x0ec6, 0x10fc, 566},
701         {0x17d7, 0x1843, 108},
702         {0x1aa7, 0x1c78, 465},
703         {0x1c79, 0x1c7d, 1},
704         {0x1d2c, 0x1d61, 1},
705         {0x1d78, 0x1d9b, 35},
706         {0x1d9c, 0x1dbf, 1},
707         {0x2071, 0x207f, 14},
708         {0x2090, 0x209c, 1},
709         {0x2c7d, 0x2d6f, 242},
710         {0x2e2f, 0x3005, 470},
711         {0x3031, 0x3035, 1},
712         {0x303b, 0x309d, 98},
713         {0x309e, 0x30fc, 94},
714         {0x30fd, 0x30fe, 1},
715         {0xa015, 0xa4f8, 1251},
716         {0xa4f9, 0xa4fd, 1},
717         {0xa60c, 0xa67f, 115},
718         {0xa717, 0xa71f, 1},
719         {0xa770, 0xa788, 24},
720         {0xa9cf, 0xaa70, 161},
721         {0xaadd, 0xff70, 21651},
722         {0xff9e, 0xff9f, 1},
723     },
724 }
725
726 var _Lo = &RangeTable{
727     R16: []Range16{
728         {0x01bb, 0x01c0, 5},
729         {0x01c1, 0x01c3, 1},
730         {0x0294, 0x05d0, 828},
731         {0x05d1, 0x05ea, 1},
732         {0x05f0, 0x05f2, 1},
733         {0x0620, 0x063f, 1},
734         {0x0641, 0x064a, 1},
735         {0x066e, 0x066f, 1},

```

736	{0x0671, 0x06d3, 1},
737	{0x06d5, 0x06ee, 25},
738	{0x06ef, 0x06fa, 11},
739	{0x06fb, 0x06fc, 1},
740	{0x06ff, 0x0710, 17},
741	{0x0712, 0x072f, 1},
742	{0x074d, 0x07a5, 1},
743	{0x07b1, 0x07ca, 25},
744	{0x07cb, 0x07ea, 1},
745	{0x0800, 0x0815, 1},
746	{0x0840, 0x0858, 1},
747	{0x0904, 0x0939, 1},
748	{0x093d, 0x0950, 19},
749	{0x0958, 0x0961, 1},
750	{0x0972, 0x0977, 1},
751	{0x0979, 0x097f, 1},
752	{0x0985, 0x098c, 1},
753	{0x098f, 0x0990, 1},
754	{0x0993, 0x09a8, 1},
755	{0x09aa, 0x09b0, 1},
756	{0x09b2, 0x09b6, 4},
757	{0x09b7, 0x09b9, 1},
758	{0x09bd, 0x09ce, 17},
759	{0x09dc, 0x09dd, 1},
760	{0x09df, 0x09e1, 1},
761	{0x09f0, 0x09f1, 1},
762	{0x0a05, 0x0a0a, 1},
763	{0x0a0f, 0x0a10, 1},
764	{0x0a13, 0x0a28, 1},
765	{0x0a2a, 0x0a30, 1},
766	{0x0a32, 0x0a33, 1},
767	{0x0a35, 0x0a36, 1},
768	{0x0a38, 0x0a39, 1},
769	{0x0a59, 0x0a5c, 1},
770	{0x0a5e, 0x0a72, 20},
771	{0x0a73, 0x0a74, 1},
772	{0x0a85, 0x0a8d, 1},
773	{0x0a8f, 0x0a91, 1},
774	{0x0a93, 0x0aa8, 1},
775	{0x0aaa, 0x0ab0, 1},
776	{0x0ab2, 0x0ab3, 1},
777	{0x0ab5, 0x0ab9, 1},
778	{0x0abd, 0x0ad0, 19},
779	{0x0ae0, 0x0ae1, 1},
780	{0x0b05, 0x0b0c, 1},
781	{0x0b0f, 0x0b10, 1},
782	{0x0b13, 0x0b28, 1},
783	{0x0b2a, 0x0b30, 1},
784	{0x0b32, 0x0b33, 1},

785	{0x0b35, 0x0b39, 1},
786	{0x0b3d, 0x0b5c, 31},
787	{0x0b5d, 0x0b5f, 2},
788	{0x0b60, 0x0b61, 1},
789	{0x0b71, 0x0b83, 18},
790	{0x0b85, 0x0b8a, 1},
791	{0x0b8e, 0x0b90, 1},
792	{0x0b92, 0x0b95, 1},
793	{0x0b99, 0x0b9a, 1},
794	{0x0b9c, 0x0b9e, 2},
795	{0x0b9f, 0x0ba3, 4},
796	{0x0ba4, 0x0ba8, 4},
797	{0x0ba9, 0x0baa, 1},
798	{0x0bae, 0x0bb9, 1},
799	{0x0bd0, 0x0c05, 53},
800	{0x0c06, 0x0c0c, 1},
801	{0x0c0e, 0x0c10, 1},
802	{0x0c12, 0x0c28, 1},
803	{0x0c2a, 0x0c33, 1},
804	{0x0c35, 0x0c39, 1},
805	{0x0c3d, 0x0c58, 27},
806	{0x0c59, 0x0c60, 7},
807	{0x0c61, 0x0c85, 36},
808	{0x0c86, 0x0c8c, 1},
809	{0x0c8e, 0x0c90, 1},
810	{0x0c92, 0x0ca8, 1},
811	{0x0caa, 0x0cb3, 1},
812	{0x0cb5, 0x0cb9, 1},
813	{0x0cbd, 0x0cde, 33},
814	{0x0ce0, 0x0ce1, 1},
815	{0x0cf1, 0x0cf2, 1},
816	{0x0d05, 0x0d0c, 1},
817	{0x0d0e, 0x0d10, 1},
818	{0x0d12, 0x0d3a, 1},
819	{0x0d3d, 0x0d4e, 17},
820	{0x0d60, 0x0d61, 1},
821	{0x0d7a, 0x0d7f, 1},
822	{0x0d85, 0x0d96, 1},
823	{0x0d9a, 0x0db1, 1},
824	{0x0db3, 0x0dbb, 1},
825	{0x0dbd, 0x0dc0, 3},
826	{0x0dc1, 0x0dc6, 1},
827	{0x0e01, 0x0e30, 1},
828	{0x0e32, 0x0e33, 1},
829	{0x0e40, 0x0e45, 1},
830	{0x0e81, 0x0e82, 1},
831	{0x0e84, 0x0e87, 3},
832	{0x0e88, 0x0e8a, 2},
833	{0x0e8d, 0x0e94, 7},
834	{0x0e95, 0x0e97, 1},

835 {0x0e99, 0x0e9f, 1},
836 {0x0ea1, 0x0ea3, 1},
837 {0x0ea5, 0x0ea7, 2},
838 {0x0eaa, 0x0eab, 1},
839 {0x0ead, 0x0eb0, 1},
840 {0x0eb2, 0x0eb3, 1},
841 {0x0ebd, 0x0ec0, 3},
842 {0x0ec1, 0x0ec4, 1},
843 {0x0edc, 0x0edd, 1},
844 {0x0f00, 0x0f40, 64},
845 {0x0f41, 0x0f47, 1},
846 {0x0f49, 0x0f6c, 1},
847 {0x0f88, 0x0f8c, 1},
848 {0x1000, 0x102a, 1},
849 {0x103f, 0x1050, 17},
850 {0x1051, 0x1055, 1},
851 {0x105a, 0x105d, 1},
852 {0x1061, 0x1065, 4},
853 {0x1066, 0x106e, 8},
854 {0x106f, 0x1070, 1},
855 {0x1075, 0x1081, 1},
856 {0x108e, 0x10d0, 66},
857 {0x10d1, 0x10fa, 1},
858 {0x1100, 0x1248, 1},
859 {0x124a, 0x124d, 1},
860 {0x1250, 0x1256, 1},
861 {0x1258, 0x125a, 2},
862 {0x125b, 0x125d, 1},
863 {0x1260, 0x1288, 1},
864 {0x128a, 0x128d, 1},
865 {0x1290, 0x12b0, 1},
866 {0x12b2, 0x12b5, 1},
867 {0x12b8, 0x12be, 1},
868 {0x12c0, 0x12c2, 2},
869 {0x12c3, 0x12c5, 1},
870 {0x12c8, 0x12d6, 1},
871 {0x12d8, 0x1310, 1},
872 {0x1312, 0x1315, 1},
873 {0x1318, 0x135a, 1},
874 {0x1380, 0x138f, 1},
875 {0x13a0, 0x13f4, 1},
876 {0x1401, 0x166c, 1},
877 {0x166f, 0x167f, 1},
878 {0x1681, 0x169a, 1},
879 {0x16a0, 0x16ea, 1},
880 {0x1700, 0x170c, 1},
881 {0x170e, 0x1711, 1},
882 {0x1720, 0x1731, 1},
883 {0x1740, 0x1751, 1},

884 {0x1760, 0x176c, 1},
885 {0x176e, 0x1770, 1},
886 {0x1780, 0x17b3, 1},
887 {0x17dc, 0x1820, 68},
888 {0x1821, 0x1842, 1},
889 {0x1844, 0x1877, 1},
890 {0x1880, 0x18a8, 1},
891 {0x18aa, 0x18b0, 6},
892 {0x18b1, 0x18f5, 1},
893 {0x1900, 0x191c, 1},
894 {0x1950, 0x196d, 1},
895 {0x1970, 0x1974, 1},
896 {0x1980, 0x19ab, 1},
897 {0x19c1, 0x19c7, 1},
898 {0x1a00, 0x1a16, 1},
899 {0x1a20, 0x1a54, 1},
900 {0x1b05, 0x1b33, 1},
901 {0x1b45, 0x1b4b, 1},
902 {0x1b83, 0x1ba0, 1},
903 {0x1bae, 0x1baf, 1},
904 {0x1bc0, 0x1be5, 1},
905 {0x1c00, 0x1c23, 1},
906 {0x1c4d, 0x1c4f, 1},
907 {0x1c5a, 0x1c77, 1},
908 {0x1ce9, 0x1cec, 1},
909 {0x1cee, 0x1cf1, 1},
910 {0x2135, 0x2138, 1},
911 {0x2d30, 0x2d65, 1},
912 {0x2d80, 0x2d96, 1},
913 {0x2da0, 0x2da6, 1},
914 {0x2da8, 0x2dae, 1},
915 {0x2db0, 0x2db6, 1},
916 {0x2db8, 0x2dbe, 1},
917 {0x2dc0, 0x2dc6, 1},
918 {0x2dc8, 0x2dce, 1},
919 {0x2dd0, 0x2dd6, 1},
920 {0x2dd8, 0x2dde, 1},
921 {0x3006, 0x303c, 54},
922 {0x3041, 0x3096, 1},
923 {0x309f, 0x30a1, 2},
924 {0x30a2, 0x30fa, 1},
925 {0x30ff, 0x3105, 6},
926 {0x3106, 0x312d, 1},
927 {0x3131, 0x318e, 1},
928 {0x31a0, 0x31ba, 1},
929 {0x31f0, 0x31ff, 1},
930 {0x3400, 0x4db5, 1},
931 {0x4e00, 0x9fcb, 1},
932 {0xa000, 0xa014, 1},

933 {0xa016, 0xa48c, 1},
934 {0xa4d0, 0xa4f7, 1},
935 {0xa500, 0xa60b, 1},
936 {0xa610, 0xa61f, 1},
937 {0xa62a, 0xa62b, 1},
938 {0xa66e, 0xa6a0, 50},
939 {0xa6a1, 0xa6e5, 1},
940 {0xa7fb, 0xa801, 1},
941 {0xa803, 0xa805, 1},
942 {0xa807, 0xa80a, 1},
943 {0xa80c, 0xa822, 1},
944 {0xa840, 0xa873, 1},
945 {0xa882, 0xa8b3, 1},
946 {0xa8f2, 0xa8f7, 1},
947 {0xa8fb, 0xa90a, 15},
948 {0xa90b, 0xa925, 1},
949 {0xa930, 0xa946, 1},
950 {0xa960, 0xa97c, 1},
951 {0xa984, 0xa9b2, 1},
952 {0xaa00, 0xaa28, 1},
953 {0xaa40, 0xaa42, 1},
954 {0xaa44, 0xaa4b, 1},
955 {0xaa60, 0xaa6f, 1},
956 {0xaa71, 0xaa76, 1},
957 {0xaa7a, 0xaa80, 6},
958 {0xaa81, 0xaaaf, 1},
959 {0xaab1, 0xaab5, 4},
960 {0xaab6, 0xaab9, 3},
961 {0xaaba, 0xaabd, 1},
962 {0xaac0, 0xaac2, 2},
963 {0xaadb, 0xaadc, 1},
964 {0xab01, 0xab06, 1},
965 {0xab09, 0xab0e, 1},
966 {0xab11, 0xab16, 1},
967 {0xab20, 0xab26, 1},
968 {0xab28, 0xab2e, 1},
969 {0xabc0, 0xabe2, 1},
970 {0xac00, 0xd7a3, 1},
971 {0xd7b0, 0xd7c6, 1},
972 {0xd7cb, 0xd7fb, 1},
973 {0xf900, 0xfa2d, 1},
974 {0xfa30, 0xfa6d, 1},
975 {0xfa70, 0xfad9, 1},
976 {0xfb1d, 0xfb1f, 2},
977 {0xfb20, 0xfb28, 1},
978 {0xfb2a, 0xfb36, 1},
979 {0xfb38, 0xfb3c, 1},
980 {0xfb3e, 0xfb40, 2},
981 {0xfb41, 0xfb43, 2},
982 {0xfb44, 0xfb46, 2},

```

983         {0xfb47, 0xfbb1, 1},
984         {0xbd3, 0xfd3d, 1},
985         {0xfd50, 0xfd8f, 1},
986         {0xfd92, 0xfdc7, 1},
987         {0xdf0, 0xdfb, 1},
988         {0xfe70, 0xfe74, 1},
989         {0xfe76, 0xfefc, 1},
990         {0xff66, 0xff6f, 1},
991         {0xff71, 0xff9d, 1},
992         {0xffa0, 0xffbe, 1},
993         {0xffc2, 0xffc7, 1},
994         {0xffca, 0xffcf, 1},
995         {0xffd2, 0xffd7, 1},
996         {0xffda, 0xffdc, 1},
997     },
998     R32: []Range32{
999         {0x10000, 0x1000b, 1},
1000        {0x1000d, 0x10026, 1},
1001        {0x10028, 0x1003a, 1},
1002        {0x1003c, 0x1003d, 1},
1003        {0x1003f, 0x1004d, 1},
1004        {0x10050, 0x1005d, 1},
1005        {0x10080, 0x100fa, 1},
1006        {0x10280, 0x1029c, 1},
1007        {0x102a0, 0x102d0, 1},
1008        {0x10300, 0x1031e, 1},
1009        {0x10330, 0x10340, 1},
1010        {0x10342, 0x10349, 1},
1011        {0x10380, 0x1039d, 1},
1012        {0x103a0, 0x103c3, 1},
1013        {0x103c8, 0x103cf, 1},
1014        {0x10450, 0x1049d, 1},
1015        {0x10800, 0x10805, 1},
1016        {0x10808, 0x1080a, 2},
1017        {0x1080b, 0x10835, 1},
1018        {0x10837, 0x10838, 1},
1019        {0x1083c, 0x1083f, 3},
1020        {0x10840, 0x10855, 1},
1021        {0x10900, 0x10915, 1},
1022        {0x10920, 0x10939, 1},
1023        {0x10a00, 0x10a10, 16},
1024        {0x10a11, 0x10a13, 1},
1025        {0x10a15, 0x10a17, 1},
1026        {0x10a19, 0x10a33, 1},
1027        {0x10a60, 0x10a7c, 1},
1028        {0x10b00, 0x10b35, 1},
1029        {0x10b40, 0x10b55, 1},
1030        {0x10b60, 0x10b72, 1},
1031        {0x10c00, 0x10c48, 1},

```

```

1032             {0x11003, 0x11037, 1},
1033             {0x11083, 0x110af, 1},
1034             {0x12000, 0x1236e, 1},
1035             {0x13000, 0x1342e, 1},
1036             {0x16800, 0x16a38, 1},
1037             {0x1b000, 0x1b001, 1},
1038             {0x20000, 0x2a6d6, 1},
1039             {0x2a700, 0x2b734, 1},
1040             {0x2b740, 0x2b81d, 1},
1041             {0x2f800, 0x2fa1d, 1},
1042         },
1043     }
1044
1045     var _Lt = &RangeTable{
1046         R16: []Range16{
1047             {0x01c5, 0x01cb, 3},
1048             {0x01f2, 0x1f88, 7574},
1049             {0x1f89, 0x1f8f, 1},
1050             {0x1f98, 0x1f9f, 1},
1051             {0x1fa8, 0x1faf, 1},
1052             {0x1fbc, 0x1fcc, 16},
1053             {0x1ffc, 0x1ffc, 1},
1054         },
1055     }
1056
1057     var _Lu = &RangeTable{
1058         R16: []Range16{
1059             {0x0041, 0x005a, 1},
1060             {0x00c0, 0x00d6, 1},
1061             {0x00d8, 0x00de, 1},
1062             {0x0100, 0x0136, 2},
1063             {0x0139, 0x0147, 2},
1064             {0x014a, 0x0178, 2},
1065             {0x0179, 0x017d, 2},
1066             {0x0181, 0x0182, 1},
1067             {0x0184, 0x0186, 2},
1068             {0x0187, 0x0189, 2},
1069             {0x018a, 0x018b, 1},
1070             {0x018e, 0x0191, 1},
1071             {0x0193, 0x0194, 1},
1072             {0x0196, 0x0198, 1},
1073             {0x019c, 0x019d, 1},
1074             {0x019f, 0x01a0, 1},
1075             {0x01a2, 0x01a6, 2},
1076             {0x01a7, 0x01a9, 2},
1077             {0x01ac, 0x01ae, 2},
1078             {0x01af, 0x01b1, 2},
1079             {0x01b2, 0x01b3, 1},
1080             {0x01b5, 0x01b7, 2},

```

1081	{0x01b8, 0x01bc, 4},
1082	{0x01c4, 0x01cd, 3},
1083	{0x01cf, 0x01db, 2},
1084	{0x01de, 0x01ee, 2},
1085	{0x01f1, 0x01f4, 3},
1086	{0x01f6, 0x01f8, 1},
1087	{0x01fa, 0x0232, 2},
1088	{0x023a, 0x023b, 1},
1089	{0x023d, 0x023e, 1},
1090	{0x0241, 0x0243, 2},
1091	{0x0244, 0x0246, 1},
1092	{0x0248, 0x024e, 2},
1093	{0x0370, 0x0372, 2},
1094	{0x0376, 0x0386, 16},
1095	{0x0388, 0x038a, 1},
1096	{0x038c, 0x038e, 2},
1097	{0x038f, 0x0391, 2},
1098	{0x0392, 0x03a1, 1},
1099	{0x03a3, 0x03ab, 1},
1100	{0x03cf, 0x03d2, 3},
1101	{0x03d3, 0x03d4, 1},
1102	{0x03d8, 0x03ee, 2},
1103	{0x03f4, 0x03f7, 3},
1104	{0x03f9, 0x03fa, 1},
1105	{0x03fd, 0x042f, 1},
1106	{0x0460, 0x0480, 2},
1107	{0x048a, 0x04c0, 2},
1108	{0x04c1, 0x04cd, 2},
1109	{0x04d0, 0x0526, 2},
1110	{0x0531, 0x0556, 1},
1111	{0x10a0, 0x10c5, 1},
1112	{0x1e00, 0x1e94, 2},
1113	{0x1e9e, 0x1efe, 2},
1114	{0x1f08, 0x1f0f, 1},
1115	{0x1f18, 0x1f1d, 1},
1116	{0x1f28, 0x1f2f, 1},
1117	{0x1f38, 0x1f3f, 1},
1118	{0x1f48, 0x1f4d, 1},
1119	{0x1f59, 0x1f5f, 2},
1120	{0x1f68, 0x1f6f, 1},
1121	{0x1fb8, 0x1fbb, 1},
1122	{0x1fc8, 0x1fcb, 1},
1123	{0x1fd8, 0x1fdb, 1},
1124	{0x1fe8, 0x1fec, 1},
1125	{0x1ff8, 0x1ffb, 1},
1126	{0x2102, 0x2107, 5},
1127	{0x210b, 0x210d, 1},
1128	{0x2110, 0x2112, 1},
1129	{0x2115, 0x2119, 4},
1130	{0x211a, 0x211d, 1},

```

1131      {0x2124, 0x212a, 2},
1132      {0x212b, 0x212d, 1},
1133      {0x2130, 0x2133, 1},
1134      {0x213e, 0x213f, 1},
1135      {0x2145, 0x2183, 62},
1136      {0x2c00, 0x2c2e, 1},
1137      {0x2c60, 0x2c62, 2},
1138      {0x2c63, 0x2c64, 1},
1139      {0x2c67, 0x2c6d, 2},
1140      {0x2c6e, 0x2c70, 1},
1141      {0x2c72, 0x2c75, 3},
1142      {0x2c7e, 0x2c80, 1},
1143      {0x2c82, 0x2ce2, 2},
1144      {0x2ceb, 0x2ced, 2},
1145      {0xa640, 0xa66c, 2},
1146      {0xa680, 0xa696, 2},
1147      {0xa722, 0xa72e, 2},
1148      {0xa732, 0xa76e, 2},
1149      {0xa779, 0xa77d, 2},
1150      {0xa77e, 0xa786, 2},
1151      {0xa78b, 0xa78d, 2},
1152      {0xa790, 0xa7a0, 16},
1153      {0xa7a2, 0xa7a8, 2},
1154      {0xff21, 0xff3a, 1},
1155      },
1156      R32: []Range32{
1157          {0x10400, 0x10427, 1},
1158          {0x1d400, 0x1d419, 1},
1159          {0x1d434, 0x1d44d, 1},
1160          {0x1d468, 0x1d481, 1},
1161          {0x1d49c, 0x1d49e, 2},
1162          {0x1d49f, 0x1d4a5, 3},
1163          {0x1d4a6, 0x1d4a9, 3},
1164          {0x1d4aa, 0x1d4ac, 1},
1165          {0x1d4ae, 0x1d4b5, 1},
1166          {0x1d4d0, 0x1d4e9, 1},
1167          {0x1d504, 0x1d505, 1},
1168          {0x1d507, 0x1d50a, 1},
1169          {0x1d50d, 0x1d514, 1},
1170          {0x1d516, 0x1d51c, 1},
1171          {0x1d538, 0x1d539, 1},
1172          {0x1d53b, 0x1d53e, 1},
1173          {0x1d540, 0x1d544, 1},
1174          {0x1d546, 0x1d54a, 4},
1175          {0x1d54b, 0x1d550, 1},
1176          {0x1d56c, 0x1d585, 1},
1177          {0x1d5a0, 0x1d5b9, 1},
1178          {0x1d5d4, 0x1d5ed, 1},
1179          {0x1d608, 0x1d621, 1},

```

```

1180         {0x1d63c, 0x1d655, 1},
1181         {0x1d670, 0x1d689, 1},
1182         {0x1d6a8, 0x1d6c0, 1},
1183         {0x1d6e2, 0x1d6fa, 1},
1184         {0x1d71c, 0x1d734, 1},
1185         {0x1d756, 0x1d76e, 1},
1186         {0x1d790, 0x1d7a8, 1},
1187         {0x1d7ca, 0x1d7ca, 1},
1188     },
1189 }
1190
1191 var _M = &RangeTable{
1192     R16: []Range16{
1193         {0x0300, 0x036f, 1},
1194         {0x0483, 0x0489, 1},
1195         {0x0591, 0x05bd, 1},
1196         {0x05bf, 0x05c1, 2},
1197         {0x05c2, 0x05c4, 2},
1198         {0x05c5, 0x05c7, 2},
1199         {0x0610, 0x061a, 1},
1200         {0x064b, 0x065f, 1},
1201         {0x0670, 0x06d6, 102},
1202         {0x06d7, 0x06dc, 1},
1203         {0x06df, 0x06e4, 1},
1204         {0x06e7, 0x06e8, 1},
1205         {0x06ea, 0x06ed, 1},
1206         {0x0711, 0x0730, 31},
1207         {0x0731, 0x074a, 1},
1208         {0x07a6, 0x07b0, 1},
1209         {0x07eb, 0x07f3, 1},
1210         {0x0816, 0x0819, 1},
1211         {0x081b, 0x0823, 1},
1212         {0x0825, 0x0827, 1},
1213         {0x0829, 0x082d, 1},
1214         {0x0859, 0x085b, 1},
1215         {0x0900, 0x0903, 1},
1216         {0x093a, 0x093c, 1},
1217         {0x093e, 0x094f, 1},
1218         {0x0951, 0x0957, 1},
1219         {0x0962, 0x0963, 1},
1220         {0x0981, 0x0983, 1},
1221         {0x09bc, 0x09be, 2},
1222         {0x09bf, 0x09c4, 1},
1223         {0x09c7, 0x09c8, 1},
1224         {0x09cb, 0x09cd, 1},
1225         {0x09d7, 0x09e2, 11},
1226         {0x09e3, 0x0a01, 30},
1227         {0x0a02, 0x0a03, 1},
1228         {0x0a3c, 0x0a3e, 2},

```

1229	{0x0a3f, 0x0a42, 1},
1230	{0x0a47, 0x0a48, 1},
1231	{0x0a4b, 0x0a4d, 1},
1232	{0x0a51, 0x0a70, 31},
1233	{0x0a71, 0x0a75, 4},
1234	{0x0a81, 0x0a83, 1},
1235	{0x0abc, 0x0abe, 2},
1236	{0x0abf, 0x0ac5, 1},
1237	{0x0ac7, 0x0ac9, 1},
1238	{0x0acb, 0x0acd, 1},
1239	{0x0ae2, 0x0ae3, 1},
1240	{0x0b01, 0x0b03, 1},
1241	{0x0b3c, 0x0b3e, 2},
1242	{0x0b3f, 0x0b44, 1},
1243	{0x0b47, 0x0b48, 1},
1244	{0x0b4b, 0x0b4d, 1},
1245	{0x0b56, 0x0b57, 1},
1246	{0x0b62, 0x0b63, 1},
1247	{0x0b82, 0x0bbe, 60},
1248	{0x0bbf, 0x0bc2, 1},
1249	{0x0bc6, 0x0bc8, 1},
1250	{0x0bca, 0x0bcd, 1},
1251	{0x0bd7, 0x0c01, 42},
1252	{0x0c02, 0x0c03, 1},
1253	{0x0c3e, 0x0c44, 1},
1254	{0x0c46, 0x0c48, 1},
1255	{0x0c4a, 0x0c4d, 1},
1256	{0x0c55, 0x0c56, 1},
1257	{0x0c62, 0x0c63, 1},
1258	{0x0c82, 0x0c83, 1},
1259	{0x0cbc, 0x0cbe, 2},
1260	{0x0cbf, 0x0cc4, 1},
1261	{0x0cc6, 0x0cc8, 1},
1262	{0x0cca, 0x0ccd, 1},
1263	{0x0cd5, 0x0cd6, 1},
1264	{0x0ce2, 0x0ce3, 1},
1265	{0x0d02, 0x0d03, 1},
1266	{0x0d3e, 0x0d44, 1},
1267	{0x0d46, 0x0d48, 1},
1268	{0x0d4a, 0x0d4d, 1},
1269	{0x0d57, 0x0d62, 11},
1270	{0x0d63, 0x0d82, 31},
1271	{0x0d83, 0x0dca, 71},
1272	{0x0dcf, 0x0dd4, 1},
1273	{0x0dd6, 0x0dd8, 2},
1274	{0x0dd9, 0x0ddf, 1},
1275	{0x0df2, 0x0df3, 1},
1276	{0x0e31, 0x0e34, 3},
1277	{0x0e35, 0x0e3a, 1},
1278	{0x0e47, 0x0e4e, 1},

1279	{0x0eb1, 0x0eb4, 3},
1280	{0x0eb5, 0x0eb9, 1},
1281	{0x0ebb, 0x0ebc, 1},
1282	{0x0ec8, 0x0ecd, 1},
1283	{0x0f18, 0x0f19, 1},
1284	{0x0f35, 0x0f39, 2},
1285	{0x0f3e, 0x0f3f, 1},
1286	{0x0f71, 0x0f84, 1},
1287	{0x0f86, 0x0f87, 1},
1288	{0x0f8d, 0x0f97, 1},
1289	{0x0f99, 0x0fbc, 1},
1290	{0x0fc6, 0x102b, 101},
1291	{0x102c, 0x103e, 1},
1292	{0x1056, 0x1059, 1},
1293	{0x105e, 0x1060, 1},
1294	{0x1062, 0x1064, 1},
1295	{0x1067, 0x106d, 1},
1296	{0x1071, 0x1074, 1},
1297	{0x1082, 0x108d, 1},
1298	{0x108f, 0x109a, 11},
1299	{0x109b, 0x109d, 1},
1300	{0x135d, 0x135f, 1},
1301	{0x1712, 0x1714, 1},
1302	{0x1732, 0x1734, 1},
1303	{0x1752, 0x1753, 1},
1304	{0x1772, 0x1773, 1},
1305	{0x17b6, 0x17d3, 1},
1306	{0x17dd, 0x180b, 46},
1307	{0x180c, 0x180d, 1},
1308	{0x18a9, 0x1920, 119},
1309	{0x1921, 0x192b, 1},
1310	{0x1930, 0x193b, 1},
1311	{0x19b0, 0x19c0, 1},
1312	{0x19c8, 0x19c9, 1},
1313	{0x1a17, 0x1a1b, 1},
1314	{0x1a55, 0x1a5e, 1},
1315	{0x1a60, 0x1a7c, 1},
1316	{0x1a7f, 0x1b00, 129},
1317	{0x1b01, 0x1b04, 1},
1318	{0x1b34, 0x1b44, 1},
1319	{0x1b6b, 0x1b73, 1},
1320	{0x1b80, 0x1b82, 1},
1321	{0x1ba1, 0x1baa, 1},
1322	{0x1be6, 0x1bf3, 1},
1323	{0x1c24, 0x1c37, 1},
1324	{0x1cd0, 0x1cd2, 1},
1325	{0x1cd4, 0x1ce8, 1},
1326	{0x1ced, 0x1cf2, 5},
1327	{0x1dc0, 0x1de6, 1},

```

1328      {0x1dfc, 0x1dff, 1},
1329      {0x20d0, 0x20f0, 1},
1330      {0x2cef, 0x2cf1, 1},
1331      {0x2d7f, 0x2de0, 97},
1332      {0x2de1, 0x2dff, 1},
1333      {0x302a, 0x302f, 1},
1334      {0x3099, 0x309a, 1},
1335      {0xa66f, 0xa672, 1},
1336      {0xa67c, 0xa67d, 1},
1337      {0xa6f0, 0xa6f1, 1},
1338      {0xa802, 0xa806, 4},
1339      {0xa80b, 0xa823, 24},
1340      {0xa824, 0xa827, 1},
1341      {0xa880, 0xa881, 1},
1342      {0xa8b4, 0xa8c4, 1},
1343      {0xa8e0, 0xa8f1, 1},
1344      {0xa926, 0xa92d, 1},
1345      {0xa947, 0xa953, 1},
1346      {0xa980, 0xa983, 1},
1347      {0xa9b3, 0xa9c0, 1},
1348      {0xaa29, 0xaa36, 1},
1349      {0xaa43, 0xaa4c, 9},
1350      {0xaa4d, 0xaa7b, 46},
1351      {0xaab0, 0xaab2, 2},
1352      {0xaab3, 0xaab4, 1},
1353      {0xaab7, 0xaab8, 1},
1354      {0xaabe, 0xaabf, 1},
1355      {0xaac1, 0xabe3, 290},
1356      {0xabe4, 0xabea, 1},
1357      {0xabec, 0xabed, 1},
1358      {0xfb1e, 0xfe00, 738},
1359      {0xfe01, 0xfe0f, 1},
1360      {0xfe20, 0xfe26, 1},
1361    },
1362    R32: []Range32{
1363      {0x101fd, 0x10a01, 2052},
1364      {0x10a02, 0x10a03, 1},
1365      {0x10a05, 0x10a06, 1},
1366      {0x10a0c, 0x10a0f, 1},
1367      {0x10a38, 0x10a3a, 1},
1368      {0x10a3f, 0x11000, 1473},
1369      {0x11001, 0x11002, 1},
1370      {0x11038, 0x11046, 1},
1371      {0x11080, 0x11082, 1},
1372      {0x110b0, 0x110ba, 1},
1373      {0x1d165, 0x1d169, 1},
1374      {0x1d16d, 0x1d172, 1},
1375      {0x1d17b, 0x1d182, 1},
1376      {0x1d185, 0x1d18b, 1},

```

```

1377             {0x1d1aa, 0x1d1ad, 1},
1378             {0x1d242, 0x1d244, 1},
1379             {0xe0100, 0xe01ef, 1},
1380         },
1381     }
1382
1383     var _Mc = &RangeTable{
1384         R16: []Range16{
1385             {0x0903, 0x093b, 56},
1386             {0x093e, 0x0940, 1},
1387             {0x0949, 0x094c, 1},
1388             {0x094e, 0x094f, 1},
1389             {0x0982, 0x0983, 1},
1390             {0x09be, 0x09c0, 1},
1391             {0x09c7, 0x09c8, 1},
1392             {0x09cb, 0x09cc, 1},
1393             {0x09d7, 0x0a03, 44},
1394             {0x0a3e, 0x0a40, 1},
1395             {0x0a83, 0x0abe, 59},
1396             {0x0abf, 0x0ac0, 1},
1397             {0x0ac9, 0x0acb, 2},
1398             {0x0acc, 0x0b02, 54},
1399             {0x0b03, 0x0b3e, 59},
1400             {0x0b40, 0x0b47, 7},
1401             {0x0b48, 0x0b4b, 3},
1402             {0x0b4c, 0x0b57, 11},
1403             {0x0bbe, 0x0bbf, 1},
1404             {0x0bc1, 0x0bc2, 1},
1405             {0x0bc6, 0x0bc8, 1},
1406             {0x0bca, 0x0bcc, 1},
1407             {0x0bd7, 0x0c01, 42},
1408             {0x0c02, 0x0c03, 1},
1409             {0x0c41, 0x0c44, 1},
1410             {0x0c82, 0x0c83, 1},
1411             {0x0cbe, 0x0cc0, 2},
1412             {0x0cc1, 0x0cc4, 1},
1413             {0x0cc7, 0x0cc8, 1},
1414             {0x0cca, 0x0ccb, 1},
1415             {0x0cd5, 0x0cd6, 1},
1416             {0x0d02, 0x0d03, 1},
1417             {0x0d3e, 0x0d40, 1},
1418             {0x0d46, 0x0d48, 1},
1419             {0x0d4a, 0x0d4c, 1},
1420             {0x0d57, 0x0d82, 43},
1421             {0x0d83, 0x0dcf, 76},
1422             {0x0dd0, 0x0dd1, 1},
1423             {0x0dd8, 0x0ddf, 1},
1424             {0x0df2, 0x0df3, 1},
1425             {0x0f3e, 0x0f3f, 1},
1426             {0x0f7f, 0x102b, 172},

```

1427	{0x102c, 0x1031, 5},
1428	{0x1038, 0x103b, 3},
1429	{0x103c, 0x1056, 26},
1430	{0x1057, 0x1062, 11},
1431	{0x1063, 0x1064, 1},
1432	{0x1067, 0x106d, 1},
1433	{0x1083, 0x1084, 1},
1434	{0x1087, 0x108c, 1},
1435	{0x108f, 0x109a, 11},
1436	{0x109b, 0x109c, 1},
1437	{0x17b6, 0x17be, 8},
1438	{0x17bf, 0x17c5, 1},
1439	{0x17c7, 0x17c8, 1},
1440	{0x1923, 0x1926, 1},
1441	{0x1929, 0x192b, 1},
1442	{0x1930, 0x1931, 1},
1443	{0x1933, 0x1938, 1},
1444	{0x19b0, 0x19c0, 1},
1445	{0x19c8, 0x19c9, 1},
1446	{0x1a19, 0x1a1b, 1},
1447	{0x1a55, 0x1a57, 2},
1448	{0x1a61, 0x1a63, 2},
1449	{0x1a64, 0x1a6d, 9},
1450	{0x1a6e, 0x1a72, 1},
1451	{0x1b04, 0x1b35, 49},
1452	{0x1b3b, 0x1b3d, 2},
1453	{0x1b3e, 0x1b41, 1},
1454	{0x1b43, 0x1b44, 1},
1455	{0x1b82, 0x1ba1, 31},
1456	{0x1ba6, 0x1ba7, 1},
1457	{0x1baa, 0x1be7, 61},
1458	{0x1bea, 0x1bec, 1},
1459	{0x1bee, 0x1bf2, 4},
1460	{0x1bf3, 0x1c24, 49},
1461	{0x1c25, 0x1c2b, 1},
1462	{0x1c34, 0x1c35, 1},
1463	{0x1ce1, 0x1cf2, 17},
1464	{0xa823, 0xa824, 1},
1465	{0xa827, 0xa880, 89},
1466	{0xa881, 0xa8b4, 51},
1467	{0xa8b5, 0xa8c3, 1},
1468	{0xa952, 0xa953, 1},
1469	{0xa983, 0xa9b4, 49},
1470	{0xa9b5, 0xa9ba, 5},
1471	{0xa9bb, 0xa9bd, 2},
1472	{0xa9be, 0xa9c0, 1},
1473	{0xaa2f, 0xaa30, 1},
1474	{0xaa33, 0xaa34, 1},
1475	{0xaa4d, 0xaa7b, 46},

```

1476             {0xabe3, 0xabe4, 1},
1477             {0xabe6, 0xabe7, 1},
1478             {0xabe9, 0xabea, 1},
1479             {0xabec, 0xabec, 1},
1480         },
1481         R32: []Range32{
1482             {0x11000, 0x11000, 1},
1483             {0x11002, 0x11082, 128},
1484             {0x110b0, 0x110b2, 1},
1485             {0x110b7, 0x110b8, 1},
1486             {0x1d165, 0x1d166, 1},
1487             {0x1d16d, 0x1d172, 1},
1488         },
1489     }
1490
1491     var _Me = &RangeTable{
1492         R16: []Range16{
1493             {0x0488, 0x0489, 1},
1494             {0x20dd, 0x20e0, 1},
1495             {0x20e2, 0x20e4, 1},
1496             {0xa670, 0xa672, 1},
1497         },
1498     }
1499
1500     var _Mn = &RangeTable{
1501         R16: []Range16{
1502             {0x0300, 0x036f, 1},
1503             {0x0483, 0x0487, 1},
1504             {0x0591, 0x05bd, 1},
1505             {0x05bf, 0x05c1, 2},
1506             {0x05c2, 0x05c4, 2},
1507             {0x05c5, 0x05c7, 2},
1508             {0x0610, 0x061a, 1},
1509             {0x064b, 0x065f, 1},
1510             {0x0670, 0x06d6, 102},
1511             {0x06d7, 0x06dc, 1},
1512             {0x06df, 0x06e4, 1},
1513             {0x06e7, 0x06e8, 1},
1514             {0x06ea, 0x06ed, 1},
1515             {0x0711, 0x0730, 31},
1516             {0x0731, 0x074a, 1},
1517             {0x07a6, 0x07b0, 1},
1518             {0x07eb, 0x07f3, 1},
1519             {0x0816, 0x0819, 1},
1520             {0x081b, 0x0823, 1},
1521             {0x0825, 0x0827, 1},
1522             {0x0829, 0x082d, 1},
1523             {0x0859, 0x085b, 1},
1524             {0x0900, 0x0902, 1},

```

1525 {0x093a, 0x093c, 2},
1526 {0x0941, 0x0948, 1},
1527 {0x094d, 0x0951, 4},
1528 {0x0952, 0x0957, 1},
1529 {0x0962, 0x0963, 1},
1530 {0x0981, 0x09bc, 59},
1531 {0x09c1, 0x09c4, 1},
1532 {0x09cd, 0x09e2, 21},
1533 {0x09e3, 0x0a01, 30},
1534 {0x0a02, 0x0a3c, 58},
1535 {0x0a41, 0x0a42, 1},
1536 {0x0a47, 0x0a48, 1},
1537 {0x0a4b, 0x0a4d, 1},
1538 {0x0a51, 0x0a70, 31},
1539 {0x0a71, 0x0a75, 4},
1540 {0x0a81, 0x0a82, 1},
1541 {0x0abc, 0x0ac1, 5},
1542 {0x0ac2, 0x0ac5, 1},
1543 {0x0ac7, 0x0ac8, 1},
1544 {0x0acd, 0x0ae2, 21},
1545 {0x0ae3, 0x0b01, 30},
1546 {0x0b3c, 0x0b3f, 3},
1547 {0x0b41, 0x0b44, 1},
1548 {0x0b4d, 0x0b56, 9},
1549 {0x0b62, 0x0b63, 1},
1550 {0x0b82, 0x0bc0, 62},
1551 {0x0bcd, 0x0c3e, 113},
1552 {0x0c3f, 0x0c40, 1},
1553 {0x0c46, 0x0c48, 1},
1554 {0x0c4a, 0x0c4d, 1},
1555 {0x0c55, 0x0c56, 1},
1556 {0x0c62, 0x0c63, 1},
1557 {0x0cbc, 0x0cbf, 3},
1558 {0x0cc6, 0x0ccc, 6},
1559 {0x0ccd, 0x0ce2, 21},
1560 {0x0ce3, 0x0d41, 94},
1561 {0x0d42, 0x0d44, 1},
1562 {0x0d4d, 0x0d62, 21},
1563 {0x0d63, 0x0dca, 103},
1564 {0x0dd2, 0x0dd4, 1},
1565 {0x0dd6, 0x0e31, 91},
1566 {0x0e34, 0x0e3a, 1},
1567 {0x0e47, 0x0e4e, 1},
1568 {0x0eb1, 0x0eb4, 3},
1569 {0x0eb5, 0x0eb9, 1},
1570 {0x0ebb, 0x0ebc, 1},
1571 {0x0ec8, 0x0ecd, 1},
1572 {0x0f18, 0x0f19, 1},
1573 {0x0f35, 0x0f39, 2},
1574 {0x0f71, 0x0f7e, 1},

1575	{0x0f80, 0x0f84, 1},
1576	{0x0f86, 0x0f87, 1},
1577	{0x0f8d, 0x0f97, 1},
1578	{0x0f99, 0x0fbc, 1},
1579	{0x0fc6, 0x102d, 103},
1580	{0x102e, 0x1030, 1},
1581	{0x1032, 0x1037, 1},
1582	{0x1039, 0x103a, 1},
1583	{0x103d, 0x103e, 1},
1584	{0x1058, 0x1059, 1},
1585	{0x105e, 0x1060, 1},
1586	{0x1071, 0x1074, 1},
1587	{0x1082, 0x1085, 3},
1588	{0x1086, 0x108d, 7},
1589	{0x109d, 0x135d, 704},
1590	{0x135e, 0x135f, 1},
1591	{0x1712, 0x1714, 1},
1592	{0x1732, 0x1734, 1},
1593	{0x1752, 0x1753, 1},
1594	{0x1772, 0x1773, 1},
1595	{0x17b7, 0x17bd, 1},
1596	{0x17c6, 0x17c9, 3},
1597	{0x17ca, 0x17d3, 1},
1598	{0x17dd, 0x180b, 46},
1599	{0x180c, 0x180d, 1},
1600	{0x18a9, 0x1920, 119},
1601	{0x1921, 0x1922, 1},
1602	{0x1927, 0x1928, 1},
1603	{0x1932, 0x1939, 7},
1604	{0x193a, 0x193b, 1},
1605	{0x1a17, 0x1a18, 1},
1606	{0x1a56, 0x1a58, 2},
1607	{0x1a59, 0x1a5e, 1},
1608	{0x1a60, 0x1a62, 2},
1609	{0x1a65, 0x1a6c, 1},
1610	{0x1a73, 0x1a7c, 1},
1611	{0x1a7f, 0x1b00, 129},
1612	{0x1b01, 0x1b03, 1},
1613	{0x1b34, 0x1b36, 2},
1614	{0x1b37, 0x1b3a, 1},
1615	{0x1b3c, 0x1b42, 6},
1616	{0x1b6b, 0x1b73, 1},
1617	{0x1b80, 0x1b81, 1},
1618	{0x1ba2, 0x1ba5, 1},
1619	{0x1ba8, 0x1ba9, 1},
1620	{0x1be6, 0x1be8, 2},
1621	{0x1be9, 0x1bed, 4},
1622	{0x1bef, 0x1bf1, 1},
1623	{0x1c2c, 0x1c33, 1},

```

1624      {0x1c36, 0x1c37, 1},
1625      {0x1cd0, 0x1cd2, 1},
1626      {0x1cd4, 0x1ce0, 1},
1627      {0x1ce2, 0x1ce8, 1},
1628      {0x1ced, 0x1dc0, 211},
1629      {0x1dc1, 0x1de6, 1},
1630      {0x1dfc, 0x1dff, 1},
1631      {0x20d0, 0x20dc, 1},
1632      {0x20e1, 0x20e5, 4},
1633      {0x20e6, 0x20f0, 1},
1634      {0x2cef, 0x2cf1, 1},
1635      {0x2d7f, 0x2de0, 97},
1636      {0x2de1, 0x2dff, 1},
1637      {0x302a, 0x302f, 1},
1638      {0x3099, 0x309a, 1},
1639      {0xa66f, 0xa67c, 13},
1640      {0xa67d, 0xa6f0, 115},
1641      {0xa6f1, 0xa802, 273},
1642      {0xa806, 0xa80b, 5},
1643      {0xa825, 0xa826, 1},
1644      {0xa8c4, 0xa8e0, 28},
1645      {0xa8e1, 0xa8f1, 1},
1646      {0xa926, 0xa92d, 1},
1647      {0xa947, 0xa951, 1},
1648      {0xa980, 0xa982, 1},
1649      {0xa9b3, 0xa9b6, 3},
1650      {0xa9b7, 0xa9b9, 1},
1651      {0xa9bc, 0xaa29, 109},
1652      {0xaa2a, 0xaa2e, 1},
1653      {0xaa31, 0xaa32, 1},
1654      {0xaa35, 0xaa36, 1},
1655      {0xaa43, 0xaa4c, 9},
1656      {0xaab0, 0xaab2, 2},
1657      {0xaab3, 0xaab4, 1},
1658      {0xaab7, 0xaab8, 1},
1659      {0xaabe, 0xaabf, 1},
1660      {0xaac1, 0xabe5, 292},
1661      {0xabe8, 0xabed, 5},
1662      {0xfb1e, 0xfe00, 738},
1663      {0xfe01, 0xfe0f, 1},
1664      {0xfe20, 0xfe26, 1},
1665      },
1666      R32: []Range32{
1667          {0x101fd, 0x10a01, 2052},
1668          {0x10a02, 0x10a03, 1},
1669          {0x10a05, 0x10a06, 1},
1670          {0x10a0c, 0x10a0f, 1},
1671          {0x10a38, 0x10a3a, 1},
1672          {0x10a3f, 0x11001, 1474},

```

```

1673             {0x11038, 0x11046, 1},
1674             {0x11080, 0x11081, 1},
1675             {0x110b3, 0x110b6, 1},
1676             {0x110b9, 0x110ba, 1},
1677             {0x1d167, 0x1d169, 1},
1678             {0x1d17b, 0x1d182, 1},
1679             {0x1d185, 0x1d18b, 1},
1680             {0x1d1aa, 0x1d1ad, 1},
1681             {0x1d242, 0x1d244, 1},
1682             {0xe0100, 0xe01ef, 1},
1683     },
1684 }
1685
1686 var _N = &RangeTable{
1687     R16: []Range16{
1688         {0x0030, 0x0039, 1},
1689         {0x00b2, 0x00b3, 1},
1690         {0x00b9, 0x00bc, 3},
1691         {0x00bd, 0x00be, 1},
1692         {0x0660, 0x0669, 1},
1693         {0x06f0, 0x06f9, 1},
1694         {0x07c0, 0x07c9, 1},
1695         {0x0966, 0x096f, 1},
1696         {0x09e6, 0x09ef, 1},
1697         {0x09f4, 0x09f9, 1},
1698         {0x0a66, 0x0a6f, 1},
1699         {0x0ae6, 0x0aef, 1},
1700         {0x0b66, 0x0b6f, 1},
1701         {0x0b72, 0x0b77, 1},
1702         {0x0be6, 0x0bf2, 1},
1703         {0x0c66, 0x0c6f, 1},
1704         {0x0c78, 0x0c7e, 1},
1705         {0x0ce6, 0x0cef, 1},
1706         {0x0d66, 0x0d75, 1},
1707         {0x0e50, 0x0e59, 1},
1708         {0x0ed0, 0x0ed9, 1},
1709         {0x0f20, 0x0f33, 1},
1710         {0x1040, 0x1049, 1},
1711         {0x1090, 0x1099, 1},
1712         {0x1369, 0x137c, 1},
1713         {0x16ee, 0x16f0, 1},
1714         {0x17e0, 0x17e9, 1},
1715         {0x17f0, 0x17f9, 1},
1716         {0x1810, 0x1819, 1},
1717         {0x1946, 0x194f, 1},
1718         {0x19d0, 0x19da, 1},
1719         {0x1a80, 0x1a89, 1},
1720         {0x1a90, 0x1a99, 1},
1721         {0x1b50, 0x1b59, 1},
1722         {0x1bb0, 0x1bb9, 1},

```

```

1723      {0x1c40, 0x1c49, 1},
1724      {0x1c50, 0x1c59, 1},
1725      {0x2070, 0x2074, 4},
1726      {0x2075, 0x2079, 1},
1727      {0x2080, 0x2089, 1},
1728      {0x2150, 0x2182, 1},
1729      {0x2185, 0x2189, 1},
1730      {0x2460, 0x249b, 1},
1731      {0x24ea, 0x24ff, 1},
1732      {0x2776, 0x2793, 1},
1733      {0x2cfd, 0x3007, 778},
1734      {0x3021, 0x3029, 1},
1735      {0x3038, 0x303a, 1},
1736      {0x3192, 0x3195, 1},
1737      {0x3220, 0x3229, 1},
1738      {0x3251, 0x325f, 1},
1739      {0x3280, 0x3289, 1},
1740      {0x32b1, 0x32bf, 1},
1741      {0xa620, 0xa629, 1},
1742      {0xa6e6, 0xa6ef, 1},
1743      {0xa830, 0xa835, 1},
1744      {0xa8d0, 0xa8d9, 1},
1745      {0xa900, 0xa909, 1},
1746      {0xa9d0, 0xa9d9, 1},
1747      {0xaa50, 0xaa59, 1},
1748      {0xabf0, 0xabf9, 1},
1749      {0xff10, 0xff19, 1},
1750      },
1751      R32: []Range32{
1752          {0x10107, 0x10133, 1},
1753          {0x10140, 0x10178, 1},
1754          {0x1018a, 0x10320, 406},
1755          {0x10321, 0x10323, 1},
1756          {0x10341, 0x1034a, 9},
1757          {0x103d1, 0x103d5, 1},
1758          {0x104a0, 0x104a9, 1},
1759          {0x10858, 0x1085f, 1},
1760          {0x10916, 0x1091b, 1},
1761          {0x10a40, 0x10a47, 1},
1762          {0x10a7d, 0x10a7e, 1},
1763          {0x10b58, 0x10b5f, 1},
1764          {0x10b78, 0x10b7f, 1},
1765          {0x10e60, 0x10e7e, 1},
1766          {0x11052, 0x1106f, 1},
1767          {0x12400, 0x12462, 1},
1768          {0x1d360, 0x1d371, 1},
1769          {0x1d7ce, 0x1d7ff, 1},
1770          {0x1f100, 0x1f10a, 1},
1771      },

```

```

1772 }
1773
1774 var _Nd = &RangeTable{
1775     R16: []Range16{
1776         {0x0030, 0x0039, 1},
1777         {0x0660, 0x0669, 1},
1778         {0x06f0, 0x06f9, 1},
1779         {0x07c0, 0x07c9, 1},
1780         {0x0966, 0x096f, 1},
1781         {0x09e6, 0x09ef, 1},
1782         {0x0a66, 0x0a6f, 1},
1783         {0x0ae6, 0x0aef, 1},
1784         {0x0b66, 0x0b6f, 1},
1785         {0x0be6, 0x0bef, 1},
1786         {0x0c66, 0x0c6f, 1},
1787         {0x0ce6, 0x0cef, 1},
1788         {0x0d66, 0x0d6f, 1},
1789         {0x0e50, 0x0e59, 1},
1790         {0x0ed0, 0x0ed9, 1},
1791         {0x0f20, 0x0f29, 1},
1792         {0x1040, 0x1049, 1},
1793         {0x1090, 0x1099, 1},
1794         {0x17e0, 0x17e9, 1},
1795         {0x1810, 0x1819, 1},
1796         {0x1946, 0x194f, 1},
1797         {0x19d0, 0x19d9, 1},
1798         {0x1a80, 0x1a89, 1},
1799         {0x1a90, 0x1a99, 1},
1800         {0x1b50, 0x1b59, 1},
1801         {0x1bb0, 0x1bb9, 1},
1802         {0x1c40, 0x1c49, 1},
1803         {0x1c50, 0x1c59, 1},
1804         {0xa620, 0xa629, 1},
1805         {0xa8d0, 0xa8d9, 1},
1806         {0xa900, 0xa909, 1},
1807         {0xa9d0, 0xa9d9, 1},
1808         {0xaa50, 0xaa59, 1},
1809         {0xabf0, 0xabf9, 1},
1810         {0xff10, 0xff19, 1},
1811     },
1812     R32: []Range32{
1813         {0x104a0, 0x104a9, 1},
1814         {0x11066, 0x1106f, 1},
1815         {0x1d7ce, 0x1d7ff, 1},
1816     },
1817 }
1818
1819 var _Nl = &RangeTable{
1820     R16: []Range16{

```

```

1821             {0x16ee, 0x16f0, 1},
1822             {0x2160, 0x2182, 1},
1823             {0x2185, 0x2188, 1},
1824             {0x3007, 0x3021, 26},
1825             {0x3022, 0x3029, 1},
1826             {0x3038, 0x303a, 1},
1827             {0xa6e6, 0xa6ef, 1},
1828         },
1829         R32: []Range32{
1830             {0x10140, 0x10174, 1},
1831             {0x10341, 0x1034a, 9},
1832             {0x103d1, 0x103d5, 1},
1833             {0x12400, 0x12462, 1},
1834         },
1835     }
1836
1837     var _No = &RangeTable{
1838         R16: []Range16{
1839             {0x00b2, 0x00b3, 1},
1840             {0x00b9, 0x00bc, 3},
1841             {0x00bd, 0x00be, 1},
1842             {0x09f4, 0x09f9, 1},
1843             {0x0b72, 0x0b77, 1},
1844             {0x0bf0, 0x0bf2, 1},
1845             {0x0c78, 0x0c7e, 1},
1846             {0x0d70, 0x0d75, 1},
1847             {0x0f2a, 0x0f33, 1},
1848             {0x1369, 0x137c, 1},
1849             {0x17f0, 0x17f9, 1},
1850             {0x19da, 0x2070, 1686},
1851             {0x2074, 0x2079, 1},
1852             {0x2080, 0x2089, 1},
1853             {0x2150, 0x215f, 1},
1854             {0x2189, 0x2460, 727},
1855             {0x2461, 0x249b, 1},
1856             {0x24ea, 0x24ff, 1},
1857             {0x2776, 0x2793, 1},
1858             {0x2cfd, 0x3192, 1173},
1859             {0x3193, 0x3195, 1},
1860             {0x3220, 0x3229, 1},
1861             {0x3251, 0x325f, 1},
1862             {0x3280, 0x3289, 1},
1863             {0x32b1, 0x32bf, 1},
1864             {0xa830, 0xa835, 1},
1865         },
1866         R32: []Range32{
1867             {0x10107, 0x10133, 1},
1868             {0x10175, 0x10178, 1},
1869             {0x1018a, 0x10320, 406},
1870             {0x10321, 0x10323, 1},

```

```

1871             {0x10858, 0x1085f, 1},
1872             {0x10916, 0x1091b, 1},
1873             {0x10a40, 0x10a47, 1},
1874             {0x10a7d, 0x10a7e, 1},
1875             {0x10b58, 0x10b5f, 1},
1876             {0x10b78, 0x10b7f, 1},
1877             {0x10e60, 0x10e7e, 1},
1878             {0x11052, 0x11065, 1},
1879             {0x1d360, 0x1d371, 1},
1880             {0x1f100, 0x1f10a, 1},
1881     },
1882 }
1883
1884 var _P = &RangeTable{
1885     R16: []Range16{
1886         {0x0021, 0x0023, 1},
1887         {0x0025, 0x002a, 1},
1888         {0x002c, 0x002f, 1},
1889         {0x003a, 0x003b, 1},
1890         {0x003f, 0x0040, 1},
1891         {0x005b, 0x005d, 1},
1892         {0x005f, 0x007b, 28},
1893         {0x007d, 0x00a1, 36},
1894         {0x00ab, 0x00b7, 12},
1895         {0x00bb, 0x00bf, 4},
1896         {0x037e, 0x0387, 9},
1897         {0x055a, 0x055f, 1},
1898         {0x0589, 0x058a, 1},
1899         {0x05be, 0x05c0, 2},
1900         {0x05c3, 0x05c6, 3},
1901         {0x05f3, 0x05f4, 1},
1902         {0x0609, 0x060a, 1},
1903         {0x060c, 0x060d, 1},
1904         {0x061b, 0x061e, 3},
1905         {0x061f, 0x066a, 75},
1906         {0x066b, 0x066d, 1},
1907         {0x06d4, 0x0700, 44},
1908         {0x0701, 0x070d, 1},
1909         {0x07f7, 0x07f9, 1},
1910         {0x0830, 0x083e, 1},
1911         {0x085e, 0x0964, 262},
1912         {0x0965, 0x0970, 11},
1913         {0x0df4, 0x0e4f, 91},
1914         {0x0e5a, 0x0e5b, 1},
1915         {0x0f04, 0x0f12, 1},
1916         {0x0f3a, 0x0f3d, 1},
1917         {0x0f85, 0x0fd0, 75},
1918         {0x0fd1, 0x0fd4, 1},
1919         {0x0fd9, 0x0fda, 1},

```

1920	{0x104a, 0x104f, 1},
1921	{0x10fb, 0x1361, 614},
1922	{0x1362, 0x1368, 1},
1923	{0x1400, 0x166d, 621},
1924	{0x166e, 0x169b, 45},
1925	{0x169c, 0x16eb, 79},
1926	{0x16ec, 0x16ed, 1},
1927	{0x1735, 0x1736, 1},
1928	{0x17d4, 0x17d6, 1},
1929	{0x17d8, 0x17da, 1},
1930	{0x1800, 0x180a, 1},
1931	{0x1944, 0x1945, 1},
1932	{0x1a1e, 0x1a1f, 1},
1933	{0x1aa0, 0x1aa6, 1},
1934	{0x1aa8, 0x1aad, 1},
1935	{0x1b5a, 0x1b60, 1},
1936	{0x1bfc, 0x1bff, 1},
1937	{0x1c3b, 0x1c3f, 1},
1938	{0x1c7e, 0x1c7f, 1},
1939	{0x1cd3, 0x2010, 829},
1940	{0x2011, 0x2027, 1},
1941	{0x2030, 0x2043, 1},
1942	{0x2045, 0x2051, 1},
1943	{0x2053, 0x205e, 1},
1944	{0x207d, 0x207e, 1},
1945	{0x208d, 0x208e, 1},
1946	{0x2329, 0x232a, 1},
1947	{0x2768, 0x2775, 1},
1948	{0x27c5, 0x27c6, 1},
1949	{0x27e6, 0x27ef, 1},
1950	{0x2983, 0x2998, 1},
1951	{0x29d8, 0x29db, 1},
1952	{0x29fc, 0x29fd, 1},
1953	{0x2cf9, 0x2cfc, 1},
1954	{0x2cfe, 0x2cff, 1},
1955	{0x2d70, 0x2e00, 144},
1956	{0x2e01, 0x2e2e, 1},
1957	{0x2e30, 0x2e31, 1},
1958	{0x3001, 0x3003, 1},
1959	{0x3008, 0x3011, 1},
1960	{0x3014, 0x301f, 1},
1961	{0x3030, 0x303d, 13},
1962	{0x30a0, 0x30fb, 91},
1963	{0xa4fe, 0xa4ff, 1},
1964	{0xa60d, 0xa60f, 1},
1965	{0xa673, 0xa67e, 11},
1966	{0xa6f2, 0xa6f7, 1},
1967	{0xa874, 0xa877, 1},
1968	{0xa8ce, 0xa8cf, 1},

```

1969         {0xa8f8, 0xa8fa, 1},
1970         {0xa92e, 0xa92f, 1},
1971         {0xa95f, 0xa9c1, 98},
1972         {0xa9c2, 0xa9cd, 1},
1973         {0xa9de, 0xa9df, 1},
1974         {0xaa5c, 0xaa5f, 1},
1975         {0xaade, 0xaadf, 1},
1976         {0xabeb, 0xfd3e, 20819},
1977         {0xfd3f, 0xfe10, 209},
1978         {0xfe11, 0xfe19, 1},
1979         {0xfe30, 0xfe52, 1},
1980         {0xfe54, 0xfe61, 1},
1981         {0xfe63, 0xfe68, 5},
1982         {0xfe6a, 0xfe6b, 1},
1983         {0xff01, 0xff03, 1},
1984         {0xff05, 0xff0a, 1},
1985         {0xff0c, 0xff0f, 1},
1986         {0xff1a, 0xff1b, 1},
1987         {0xff1f, 0xff20, 1},
1988         {0xff3b, 0xff3d, 1},
1989         {0xff3f, 0xff5b, 28},
1990         {0xff5d, 0xff5f, 2},
1991         {0xff60, 0xff65, 1},
1992     },
1993     R32: []Range32{
1994         {0x10100, 0x10101, 1},
1995         {0x1039f, 0x103d0, 49},
1996         {0x10857, 0x1091f, 200},
1997         {0x1093f, 0x10a50, 273},
1998         {0x10a51, 0x10a58, 1},
1999         {0x10a7f, 0x10b39, 186},
2000         {0x10b3a, 0x10b3f, 1},
2001         {0x11047, 0x1104d, 1},
2002         {0x110bb, 0x110bc, 1},
2003         {0x110be, 0x110c1, 1},
2004         {0x12470, 0x12473, 1},
2005     },
2006 }
2007
2008 var _Pc = &RangeTable{
2009     R16: []Range16{
2010         {0x005f, 0x203f, 8160},
2011         {0x2040, 0x2054, 20},
2012         {0xfe33, 0xfe34, 1},
2013         {0xfe4d, 0xfe4f, 1},
2014         {0xff3f, 0xff3f, 1},
2015     },
2016 }
2017
2018 var _Pd = &RangeTable{

```

```

2019         R16: []Range16{
2020             {0x002d, 0x058a, 1373},
2021             {0x05be, 0x1400, 3650},
2022             {0x1806, 0x2010, 2058},
2023             {0x2011, 0x2015, 1},
2024             {0x2e17, 0x2e1a, 3},
2025             {0x301c, 0x3030, 20},
2026             {0x30a0, 0xfe31, 52625},
2027             {0xfe32, 0xfe58, 38},
2028             {0xfe63, 0xff0d, 170},
2029         },
2030     }
2031
2032     var _Pe = &RangeTable{
2033         R16: []Range16{
2034             {0x0029, 0x005d, 52},
2035             {0x007d, 0x0f3b, 3774},
2036             {0x0f3d, 0x169c, 1887},
2037             {0x2046, 0x207e, 56},
2038             {0x208e, 0x232a, 668},
2039             {0x2769, 0x2775, 2},
2040             {0x27c6, 0x27e7, 33},
2041             {0x27e9, 0x27ef, 2},
2042             {0x2984, 0x2998, 2},
2043             {0x29d9, 0x29db, 2},
2044             {0x29fd, 0x2e23, 1062},
2045             {0x2e25, 0x2e29, 2},
2046             {0x3009, 0x3011, 2},
2047             {0x3015, 0x301b, 2},
2048             {0x301e, 0x301f, 1},
2049             {0xfd3f, 0xfe18, 217},
2050             {0xfe36, 0xfe44, 2},
2051             {0xfe48, 0xfe5a, 18},
2052             {0xfe5c, 0xfe5e, 2},
2053             {0xff09, 0xff3d, 52},
2054             {0xff5d, 0xff63, 3},
2055         },
2056     }
2057
2058     var _Pf = &RangeTable{
2059         R16: []Range16{
2060             {0x00bb, 0x2019, 8030},
2061             {0x201d, 0x203a, 29},
2062             {0x2e03, 0x2e05, 2},
2063             {0x2e0a, 0x2e0d, 3},
2064             {0x2e1d, 0x2e21, 4},
2065         },
2066     }
2067

```

```

2068 var _Pi = &RangeTable{
2069     R16: []Range16{
2070         {0x00ab, 0x2018, 8045},
2071         {0x201b, 0x201c, 1},
2072         {0x201f, 0x2039, 26},
2073         {0x2e02, 0x2e04, 2},
2074         {0x2e09, 0x2e0c, 3},
2075         {0x2e1c, 0x2e20, 4},
2076     },
2077 }
2078
2079 var _Po = &RangeTable{
2080     R16: []Range16{
2081         {0x0021, 0x0023, 1},
2082         {0x0025, 0x0027, 1},
2083         {0x002a, 0x002e, 2},
2084         {0x002f, 0x003a, 11},
2085         {0x003b, 0x003f, 4},
2086         {0x0040, 0x005c, 28},
2087         {0x00a1, 0x00b7, 22},
2088         {0x00bf, 0x037e, 703},
2089         {0x0387, 0x055a, 467},
2090         {0x055b, 0x055f, 1},
2091         {0x0589, 0x05c0, 55},
2092         {0x05c3, 0x05c6, 3},
2093         {0x05f3, 0x05f4, 1},
2094         {0x0609, 0x060a, 1},
2095         {0x060c, 0x060d, 1},
2096         {0x061b, 0x061e, 3},
2097         {0x061f, 0x066a, 75},
2098         {0x066b, 0x066d, 1},
2099         {0x06d4, 0x0700, 44},
2100         {0x0701, 0x070d, 1},
2101         {0x07f7, 0x07f9, 1},
2102         {0x0830, 0x083e, 1},
2103         {0x085e, 0x0964, 262},
2104         {0x0965, 0x0970, 11},
2105         {0x0df4, 0x0e4f, 91},
2106         {0x0e5a, 0x0e5b, 1},
2107         {0x0f04, 0x0f12, 1},
2108         {0x0f85, 0x0fd0, 75},
2109         {0x0fd1, 0x0fd4, 1},
2110         {0x0fd9, 0x0fda, 1},
2111         {0x104a, 0x104f, 1},
2112         {0x10fb, 0x1361, 614},
2113         {0x1362, 0x1368, 1},
2114         {0x166d, 0x166e, 1},
2115         {0x16eb, 0x16ed, 1},
2116         {0x1735, 0x1736, 1},

```

2117	{0x17d4, 0x17d6, 1},
2118	{0x17d8, 0x17da, 1},
2119	{0x1800, 0x1805, 1},
2120	{0x1807, 0x180a, 1},
2121	{0x1944, 0x1945, 1},
2122	{0x1a1e, 0x1a1f, 1},
2123	{0x1aa0, 0x1aa6, 1},
2124	{0x1aa8, 0x1aad, 1},
2125	{0x1b5a, 0x1b60, 1},
2126	{0x1bfc, 0x1bff, 1},
2127	{0x1c3b, 0x1c3f, 1},
2128	{0x1c7e, 0x1c7f, 1},
2129	{0x1cd3, 0x2016, 835},
2130	{0x2017, 0x2020, 9},
2131	{0x2021, 0x2027, 1},
2132	{0x2030, 0x2038, 1},
2133	{0x203b, 0x203e, 1},
2134	{0x2041, 0x2043, 1},
2135	{0x2047, 0x2051, 1},
2136	{0x2053, 0x2055, 2},
2137	{0x2056, 0x205e, 1},
2138	{0x2cf9, 0x2cfc, 1},
2139	{0x2cfe, 0x2cff, 1},
2140	{0x2d70, 0x2e00, 144},
2141	{0x2e01, 0x2e06, 5},
2142	{0x2e07, 0x2e08, 1},
2143	{0x2e0b, 0x2e0e, 3},
2144	{0x2e0f, 0x2e16, 1},
2145	{0x2e18, 0x2e19, 1},
2146	{0x2e1b, 0x2e1e, 3},
2147	{0x2e1f, 0x2e2a, 11},
2148	{0x2e2b, 0x2e2e, 1},
2149	{0x2e30, 0x2e31, 1},
2150	{0x3001, 0x3003, 1},
2151	{0x303d, 0x30fb, 190},
2152	{0xa4fe, 0xa4ff, 1},
2153	{0xa60d, 0xa60f, 1},
2154	{0xa673, 0xa67e, 11},
2155	{0xa6f2, 0xa6f7, 1},
2156	{0xa874, 0xa877, 1},
2157	{0xa8ce, 0xa8cf, 1},
2158	{0xa8f8, 0xa8fa, 1},
2159	{0xa92e, 0xa92f, 1},
2160	{0xa95f, 0xa9c1, 98},
2161	{0xa9c2, 0xa9cd, 1},
2162	{0xa9de, 0xa9df, 1},
2163	{0xaa5c, 0xaa5f, 1},
2164	{0xaade, 0xaadf, 1},
2165	{0xabeb, 0xfe10, 21029},
2166	{0xfe11, 0xfe16, 1},

```

2167         {0xfe19, 0xfe30, 23},
2168         {0xfe45, 0xfe46, 1},
2169         {0xfe49, 0xfe4c, 1},
2170         {0xfe50, 0xfe52, 1},
2171         {0xfe54, 0xfe57, 1},
2172         {0xfe5f, 0xfe61, 1},
2173         {0xfe68, 0xfe6a, 2},
2174         {0xfe6b, 0xff01, 150},
2175         {0xff02, 0xff03, 1},
2176         {0xff05, 0xff07, 1},
2177         {0xff0a, 0xff0e, 2},
2178         {0xff0f, 0xff1a, 11},
2179         {0xff1b, 0xff1f, 4},
2180         {0xff20, 0xff3c, 28},
2181         {0xff61, 0xff64, 3},
2182         {0xff65, 0xff65, 1},
2183     },
2184     R32: []Range32{
2185         {0x10100, 0x10100, 1},
2186         {0x10101, 0x1039f, 670},
2187         {0x103d0, 0x10857, 1159},
2188         {0x1091f, 0x1093f, 32},
2189         {0x10a50, 0x10a58, 1},
2190         {0x10a7f, 0x10b39, 186},
2191         {0x10b3a, 0x10b3f, 1},
2192         {0x11047, 0x1104d, 1},
2193         {0x110bb, 0x110bc, 1},
2194         {0x110be, 0x110c1, 1},
2195         {0x12470, 0x12473, 1},
2196     },
2197 }
2198
2199 var _Ps = &RangeTable{
2200     R16: []Range16{
2201         {0x0028, 0x005b, 51},
2202         {0x007b, 0x0f3a, 3775},
2203         {0x0f3c, 0x169b, 1887},
2204         {0x201a, 0x201e, 4},
2205         {0x2045, 0x207d, 56},
2206         {0x208d, 0x2329, 668},
2207         {0x2768, 0x2774, 2},
2208         {0x27c5, 0x27e6, 33},
2209         {0x27e8, 0x27ee, 2},
2210         {0x2983, 0x2997, 2},
2211         {0x29d8, 0x29da, 2},
2212         {0x29fc, 0x2e22, 1062},
2213         {0x2e24, 0x2e28, 2},
2214         {0x3008, 0x3010, 2},
2215         {0x3014, 0x301a, 2},

```

```

2216             {0x301d, 0xfd3e, 52513},
2217             {0xfe17, 0xfe35, 30},
2218             {0xfe37, 0xfe43, 2},
2219             {0xfe47, 0xfe59, 18},
2220             {0xfe5b, 0xfe5d, 2},
2221             {0xff08, 0xff3b, 51},
2222             {0xff5b, 0xff5f, 4},
2223             {0xff62, 0xff62, 1},
2224         },
2225     }
2226
2227     var _S = &RangeTable{
2228         R16: []Range16{
2229             {0x0024, 0x002b, 7},
2230             {0x003c, 0x003e, 1},
2231             {0x005e, 0x0060, 2},
2232             {0x007c, 0x007e, 2},
2233             {0x00a2, 0x00a9, 1},
2234             {0x00ac, 0x00ae, 2},
2235             {0x00af, 0x00b1, 1},
2236             {0x00b4, 0x00b8, 2},
2237             {0x00d7, 0x00f7, 32},
2238             {0x02c2, 0x02c5, 1},
2239             {0x02d2, 0x02df, 1},
2240             {0x02e5, 0x02eb, 1},
2241             {0x02ed, 0x02ef, 2},
2242             {0x02f0, 0x02ff, 1},
2243             {0x0375, 0x0384, 15},
2244             {0x0385, 0x03f6, 113},
2245             {0x0482, 0x0606, 388},
2246             {0x0607, 0x0608, 1},
2247             {0x060b, 0x060e, 3},
2248             {0x060f, 0x06de, 207},
2249             {0x06e9, 0x06fd, 20},
2250             {0x06fe, 0x07f6, 248},
2251             {0x09f2, 0x09f3, 1},
2252             {0x09fa, 0x09fb, 1},
2253             {0x0af1, 0x0b70, 127},
2254             {0x0bf3, 0x0bfa, 1},
2255             {0x0c7f, 0x0d79, 250},
2256             {0x0e3f, 0x0f01, 194},
2257             {0x0f02, 0x0f03, 1},
2258             {0x0f13, 0x0f17, 1},
2259             {0x0f1a, 0x0f1f, 1},
2260             {0x0f34, 0x0f38, 2},
2261             {0x0fbe, 0x0fc5, 1},
2262             {0x0fc7, 0x0fcc, 1},
2263             {0x0fce, 0x0fcf, 1},
2264             {0x0fd5, 0x0fd8, 1},

```

2265	{0x109e, 0x109f, 1},
2266	{0x1360, 0x1390, 48},
2267	{0x1391, 0x1399, 1},
2268	{0x17db, 0x1940, 357},
2269	{0x19de, 0x19ff, 1},
2270	{0x1b61, 0x1b6a, 1},
2271	{0x1b74, 0x1b7c, 1},
2272	{0x1fbd, 0x1fbf, 2},
2273	{0x1fc0, 0x1fc1, 1},
2274	{0x1fcd, 0x1fcf, 1},
2275	{0x1fdd, 0x1fdf, 1},
2276	{0x1fed, 0x1fef, 1},
2277	{0x1ffd, 0x1ffe, 1},
2278	{0x2044, 0x2052, 14},
2279	{0x207a, 0x207c, 1},
2280	{0x208a, 0x208c, 1},
2281	{0x20a0, 0x20b9, 1},
2282	{0x2100, 0x2101, 1},
2283	{0x2103, 0x2106, 1},
2284	{0x2108, 0x2109, 1},
2285	{0x2114, 0x2116, 2},
2286	{0x2117, 0x2118, 1},
2287	{0x211e, 0x2123, 1},
2288	{0x2125, 0x2129, 2},
2289	{0x212e, 0x213a, 12},
2290	{0x213b, 0x2140, 5},
2291	{0x2141, 0x2144, 1},
2292	{0x214a, 0x214d, 1},
2293	{0x214f, 0x2190, 65},
2294	{0x2191, 0x2328, 1},
2295	{0x232b, 0x23f3, 1},
2296	{0x2400, 0x2426, 1},
2297	{0x2440, 0x244a, 1},
2298	{0x249c, 0x24e9, 1},
2299	{0x2500, 0x26ff, 1},
2300	{0x2701, 0x2767, 1},
2301	{0x2794, 0x27c4, 1},
2302	{0x27c7, 0x27ca, 1},
2303	{0x27cc, 0x27ce, 2},
2304	{0x27cf, 0x27e5, 1},
2305	{0x27f0, 0x2982, 1},
2306	{0x2999, 0x29d7, 1},
2307	{0x29dc, 0x29fb, 1},
2308	{0x29fe, 0x2b4c, 1},
2309	{0x2b50, 0x2b59, 1},
2310	{0x2ce5, 0x2cea, 1},
2311	{0x2e80, 0x2e99, 1},
2312	{0x2e9b, 0x2ef3, 1},
2313	{0x2f00, 0x2fd5, 1},
2314	{0x2ff0, 0x2ffb, 1},

```

2315      {0x3004, 0x3012, 14},
2316      {0x3013, 0x3020, 13},
2317      {0x3036, 0x3037, 1},
2318      {0x303e, 0x303f, 1},
2319      {0x309b, 0x309c, 1},
2320      {0x3190, 0x3191, 1},
2321      {0x3196, 0x319f, 1},
2322      {0x31c0, 0x31e3, 1},
2323      {0x3200, 0x321e, 1},
2324      {0x322a, 0x3250, 1},
2325      {0x3260, 0x327f, 1},
2326      {0x328a, 0x32b0, 1},
2327      {0x32c0, 0x32fe, 1},
2328      {0x3300, 0x33ff, 1},
2329      {0x4dc0, 0x4dff, 1},
2330      {0xa490, 0xa4c6, 1},
2331      {0xa700, 0xa716, 1},
2332      {0xa720, 0xa721, 1},
2333      {0xa789, 0xa78a, 1},
2334      {0xa828, 0xa82b, 1},
2335      {0xa836, 0xa839, 1},
2336      {0xaa77, 0xaa79, 1},
2337      {0xfb29, 0xfbb2, 137},
2338      {0xfbb3, 0xfbc1, 1},
2339      {0xfdfe, 0xfdfe, 1},
2340      {0xfe62, 0xfe64, 2},
2341      {0xfe65, 0xfe66, 1},
2342      {0xfe69, 0xff04, 155},
2343      {0xff0b, 0xff1c, 17},
2344      {0xff1d, 0xff1e, 1},
2345      {0xff3e, 0xff40, 2},
2346      {0xff5c, 0xff5e, 2},
2347      {0xffe0, 0xffe6, 1},
2348      {0xffe8, 0xffee, 1},
2349      {0xfffc, 0xfffd, 1},
2350      },
2351      R32: []Range32{
2352          {0x10102, 0x10137, 53},
2353          {0x10138, 0x1013f, 1},
2354          {0x10179, 0x10189, 1},
2355          {0x10190, 0x1019b, 1},
2356          {0x101d0, 0x101fc, 1},
2357          {0x1d000, 0x1d0f5, 1},
2358          {0x1d100, 0x1d126, 1},
2359          {0x1d129, 0x1d164, 1},
2360          {0x1d16a, 0x1d16c, 1},
2361          {0x1d183, 0x1d184, 1},
2362          {0x1d18c, 0x1d1a9, 1},
2363          {0x1d1ae, 0x1d1dd, 1},

```

```
2364 {0x1d200, 0x1d241, 1},
2365 {0x1d245, 0x1d300, 187},
2366 {0x1d301, 0x1d356, 1},
2367 {0x1d6c1, 0x1d6db, 26},
2368 {0x1d6fb, 0x1d715, 26},
2369 {0x1d735, 0x1d74f, 26},
2370 {0x1d76f, 0x1d789, 26},
2371 {0x1d7a9, 0x1d7c3, 26},
2372 {0x1f000, 0x1f02b, 1},
2373 {0x1f030, 0x1f093, 1},
2374 {0x1f0a0, 0x1f0ae, 1},
2375 {0x1f0b1, 0x1f0be, 1},
2376 {0x1f0c1, 0x1f0cf, 1},
2377 {0x1f0d1, 0x1f0df, 1},
2378 {0x1f110, 0x1f12e, 1},
2379 {0x1f130, 0x1f169, 1},
2380 {0x1f170, 0x1f19a, 1},
2381 {0x1f1e6, 0x1f202, 1},
2382 {0x1f210, 0x1f23a, 1},
2383 {0x1f240, 0x1f248, 1},
2384 {0x1f250, 0x1f251, 1},
2385 {0x1f300, 0x1f320, 1},
2386 {0x1f330, 0x1f335, 1},
2387 {0x1f337, 0x1f37c, 1},
2388 {0x1f380, 0x1f393, 1},
2389 {0x1f3a0, 0x1f3c4, 1},
2390 {0x1f3c6, 0x1f3ca, 1},
2391 {0x1f3e0, 0x1f3f0, 1},
2392 {0x1f400, 0x1f43e, 1},
2393 {0x1f440, 0x1f442, 2},
2394 {0x1f443, 0x1f4f7, 1},
2395 {0x1f4f9, 0x1f4fc, 1},
2396 {0x1f500, 0x1f53d, 1},
2397 {0x1f550, 0x1f567, 1},
2398 {0x1f5fb, 0x1f5ff, 1},
2399 {0x1f601, 0x1f610, 1},
2400 {0x1f612, 0x1f614, 1},
2401 {0x1f616, 0x1f61c, 2},
2402 {0x1f61d, 0x1f61e, 1},
2403 {0x1f620, 0x1f625, 1},
2404 {0x1f628, 0x1f62b, 1},
2405 {0x1f62d, 0x1f630, 3},
2406 {0x1f631, 0x1f633, 1},
2407 {0x1f635, 0x1f640, 1},
2408 {0x1f645, 0x1f64f, 1},
2409 {0x1f680, 0x1f6c5, 1},
2410 {0x1f700, 0x1f773, 1},
2411 },
2412 }
```

```

2413
2414 var _Sc = &RangeTable{
2415     R16: []Range16{
2416         {0x0024, 0x00a2, 126},
2417         {0x00a3, 0x00a5, 1},
2418         {0x060b, 0x09f2, 999},
2419         {0x09f3, 0x09fb, 8},
2420         {0x0af1, 0x0bf9, 264},
2421         {0x0e3f, 0x17db, 2460},
2422         {0x20a0, 0x20b9, 1},
2423         {0xa838, 0xfdfc, 21956},
2424         {0xfe69, 0xff04, 155},
2425         {0xffe0, 0xffe1, 1},
2426         {0xffe5, 0xffe6, 1},
2427     },
2428 }
2429
2430 var _Sk = &RangeTable{
2431     R16: []Range16{
2432         {0x005e, 0x0060, 2},
2433         {0x00a8, 0x00af, 7},
2434         {0x00b4, 0x00b8, 4},
2435         {0x02c2, 0x02c5, 1},
2436         {0x02d2, 0x02df, 1},
2437         {0x02e5, 0x02eb, 1},
2438         {0x02ed, 0x02ef, 2},
2439         {0x02f0, 0x02ff, 1},
2440         {0x0375, 0x0384, 15},
2441         {0x0385, 0x1fbd, 7224},
2442         {0x1fbf, 0x1fc1, 1},
2443         {0x1fcd, 0x1fcf, 1},
2444         {0x1fdd, 0x1fdf, 1},
2445         {0x1fed, 0x1fef, 1},
2446         {0x1ffd, 0x1ffe, 1},
2447         {0x309b, 0x309c, 1},
2448         {0xa700, 0xa716, 1},
2449         {0xa720, 0xa721, 1},
2450         {0xa789, 0xa78a, 1},
2451         {0xfbb2, 0xfbc1, 1},
2452         {0xff3e, 0xff40, 2},
2453         {0xffe3, 0xffe3, 1},
2454     },
2455 }
2456
2457 var _Sm = &RangeTable{
2458     R16: []Range16{
2459         {0x002b, 0x003c, 17},
2460         {0x003d, 0x003e, 1},
2461         {0x007c, 0x007e, 2},
2462         {0x00ac, 0x00b1, 5},

```

```

2463      {0x00d7, 0x00f7, 32},
2464      {0x03f6, 0x0606, 528},
2465      {0x0607, 0x0608, 1},
2466      {0x2044, 0x2052, 14},
2467      {0x207a, 0x207c, 1},
2468      {0x208a, 0x208c, 1},
2469      {0x2118, 0x2140, 40},
2470      {0x2141, 0x2144, 1},
2471      {0x214b, 0x2190, 69},
2472      {0x2191, 0x2194, 1},
2473      {0x219a, 0x219b, 1},
2474      {0x21a0, 0x21a6, 3},
2475      {0x21ae, 0x21ce, 32},
2476      {0x21cf, 0x21d2, 3},
2477      {0x21d4, 0x21f4, 32},
2478      {0x21f5, 0x22ff, 1},
2479      {0x2308, 0x230b, 1},
2480      {0x2320, 0x2321, 1},
2481      {0x237c, 0x239b, 31},
2482      {0x239c, 0x23b3, 1},
2483      {0x23dc, 0x23e1, 1},
2484      {0x25b7, 0x25c1, 10},
2485      {0x25f8, 0x25ff, 1},
2486      {0x266f, 0x27c0, 337},
2487      {0x27c1, 0x27c4, 1},
2488      {0x27c7, 0x27ca, 1},
2489      {0x27cc, 0x27ce, 2},
2490      {0x27cf, 0x27e5, 1},
2491      {0x27f0, 0x27ff, 1},
2492      {0x2900, 0x2982, 1},
2493      {0x2999, 0x29d7, 1},
2494      {0x29dc, 0x29fb, 1},
2495      {0x29fe, 0x2aff, 1},
2496      {0x2b30, 0x2b44, 1},
2497      {0x2b47, 0x2b4c, 1},
2498      {0xfb29, 0xfe62, 825},
2499      {0xfe64, 0xfe66, 1},
2500      {0xff0b, 0xff1c, 17},
2501      {0xff1d, 0xff1e, 1},
2502      {0xff5c, 0xff5e, 2},
2503      {0xffe2, 0xffe9, 7},
2504      {0xffea, 0xffec, 1},
2505      },
2506      R32: []Range32{
2507          {0x1d6c1, 0x1d6db, 26},
2508          {0x1d6fb, 0x1d715, 26},
2509          {0x1d735, 0x1d74f, 26},
2510          {0x1d76f, 0x1d789, 26},
2511          {0x1d7a9, 0x1d7c3, 26},

```

```

2512     },
2513 }
2514
2515 var _So = &RangeTable{
2516     R16: []Range16{
2517         {0x00a6, 0x00a7, 1},
2518         {0x00a9, 0x00ae, 5},
2519         {0x00b0, 0x00b6, 6},
2520         {0x0482, 0x060e, 396},
2521         {0x060f, 0x06de, 207},
2522         {0x06e9, 0x06fd, 20},
2523         {0x06fe, 0x07f6, 248},
2524         {0x09fa, 0x0b70, 374},
2525         {0x0bf3, 0x0bf8, 1},
2526         {0x0bfa, 0x0c7f, 133},
2527         {0x0d79, 0x0f01, 392},
2528         {0x0f02, 0x0f03, 1},
2529         {0x0f13, 0x0f17, 1},
2530         {0x0f1a, 0x0f1f, 1},
2531         {0x0f34, 0x0f38, 2},
2532         {0x0fbe, 0x0fc5, 1},
2533         {0x0fc7, 0x0fcc, 1},
2534         {0x0fce, 0x0fcf, 1},
2535         {0x0fd5, 0x0fd8, 1},
2536         {0x109e, 0x109f, 1},
2537         {0x1360, 0x1390, 48},
2538         {0x1391, 0x1399, 1},
2539         {0x1940, 0x19de, 158},
2540         {0x19df, 0x19ff, 1},
2541         {0x1b61, 0x1b6a, 1},
2542         {0x1b74, 0x1b7c, 1},
2543         {0x2100, 0x2101, 1},
2544         {0x2103, 0x2106, 1},
2545         {0x2108, 0x2109, 1},
2546         {0x2114, 0x2116, 2},
2547         {0x2117, 0x211e, 7},
2548         {0x211f, 0x2123, 1},
2549         {0x2125, 0x2129, 2},
2550         {0x212e, 0x213a, 12},
2551         {0x213b, 0x214a, 15},
2552         {0x214c, 0x214d, 1},
2553         {0x214f, 0x2195, 70},
2554         {0x2196, 0x2199, 1},
2555         {0x219c, 0x219f, 1},
2556         {0x21a1, 0x21a2, 1},
2557         {0x21a4, 0x21a5, 1},
2558         {0x21a7, 0x21ad, 1},
2559         {0x21af, 0x21cd, 1},
2560         {0x21d0, 0x21d1, 1},

```

2561	{0x21d3, 0x21d5, 2},
2562	{0x21d6, 0x21f3, 1},
2563	{0x2300, 0x2307, 1},
2564	{0x230c, 0x231f, 1},
2565	{0x2322, 0x2328, 1},
2566	{0x232b, 0x237b, 1},
2567	{0x237d, 0x239a, 1},
2568	{0x23b4, 0x23db, 1},
2569	{0x23e2, 0x23f3, 1},
2570	{0x2400, 0x2426, 1},
2571	{0x2440, 0x244a, 1},
2572	{0x249c, 0x24e9, 1},
2573	{0x2500, 0x25b6, 1},
2574	{0x25b8, 0x25c0, 1},
2575	{0x25c2, 0x25f7, 1},
2576	{0x2600, 0x266e, 1},
2577	{0x2670, 0x26ff, 1},
2578	{0x2701, 0x2767, 1},
2579	{0x2794, 0x27bf, 1},
2580	{0x2800, 0x28ff, 1},
2581	{0x2b00, 0x2b2f, 1},
2582	{0x2b45, 0x2b46, 1},
2583	{0x2b50, 0x2b59, 1},
2584	{0x2ce5, 0x2cea, 1},
2585	{0x2e80, 0x2e99, 1},
2586	{0x2e9b, 0x2ef3, 1},
2587	{0x2f00, 0x2fd5, 1},
2588	{0x2ff0, 0x2ffb, 1},
2589	{0x3004, 0x3012, 14},
2590	{0x3013, 0x3020, 13},
2591	{0x3036, 0x3037, 1},
2592	{0x303e, 0x303f, 1},
2593	{0x3190, 0x3191, 1},
2594	{0x3196, 0x319f, 1},
2595	{0x31c0, 0x31e3, 1},
2596	{0x3200, 0x321e, 1},
2597	{0x322a, 0x3250, 1},
2598	{0x3260, 0x327f, 1},
2599	{0x328a, 0x32b0, 1},
2600	{0x32c0, 0x32fe, 1},
2601	{0x3300, 0x33ff, 1},
2602	{0x4dc0, 0x4dff, 1},
2603	{0xa490, 0xa4c6, 1},
2604	{0xa828, 0xa82b, 1},
2605	{0xa836, 0xa837, 1},
2606	{0xa839, 0xaa77, 574},
2607	{0xaa78, 0xaa79, 1},
2608	{0xfdfd, 0xffe4, 487},
2609	{0xffe8, 0xffed, 5},
2610	{0xffee, 0xfffc, 14},

```

2611         {0xffffd, 0xffffd, 1},
2612     },
2613     R32: []Range32{
2614         {0x10102, 0x10102, 1},
2615         {0x10137, 0x1013f, 1},
2616         {0x10179, 0x10189, 1},
2617         {0x10190, 0x1019b, 1},
2618         {0x101d0, 0x101fc, 1},
2619         {0x1d000, 0x1d0f5, 1},
2620         {0x1d100, 0x1d126, 1},
2621         {0x1d129, 0x1d164, 1},
2622         {0x1d16a, 0x1d16c, 1},
2623         {0x1d183, 0x1d184, 1},
2624         {0x1d18c, 0x1d1a9, 1},
2625         {0x1d1ae, 0x1d1dd, 1},
2626         {0x1d200, 0x1d241, 1},
2627         {0x1d245, 0x1d300, 187},
2628         {0x1d301, 0x1d356, 1},
2629         {0x1f000, 0x1f02b, 1},
2630         {0x1f030, 0x1f093, 1},
2631         {0x1f0a0, 0x1f0ae, 1},
2632         {0x1f0b1, 0x1f0be, 1},
2633         {0x1f0c1, 0x1f0cf, 1},
2634         {0x1f0d1, 0x1f0df, 1},
2635         {0x1f110, 0x1f12e, 1},
2636         {0x1f130, 0x1f169, 1},
2637         {0x1f170, 0x1f19a, 1},
2638         {0x1f1e6, 0x1f202, 1},
2639         {0x1f210, 0x1f23a, 1},
2640         {0x1f240, 0x1f248, 1},
2641         {0x1f250, 0x1f251, 1},
2642         {0x1f300, 0x1f320, 1},
2643         {0x1f330, 0x1f335, 1},
2644         {0x1f337, 0x1f37c, 1},
2645         {0x1f380, 0x1f393, 1},
2646         {0x1f3a0, 0x1f3c4, 1},
2647         {0x1f3c6, 0x1f3ca, 1},
2648         {0x1f3e0, 0x1f3f0, 1},
2649         {0x1f400, 0x1f43e, 1},
2650         {0x1f440, 0x1f442, 2},
2651         {0x1f443, 0x1f4f7, 1},
2652         {0x1f4f9, 0x1f4fc, 1},
2653         {0x1f500, 0x1f53d, 1},
2654         {0x1f550, 0x1f567, 1},
2655         {0x1f5fb, 0x1f5ff, 1},
2656         {0x1f601, 0x1f610, 1},
2657         {0x1f612, 0x1f614, 1},
2658         {0x1f616, 0x1f61c, 2},
2659         {0x1f61d, 0x1f61e, 1},

```

```

2660             {0x1f620, 0x1f625, 1},
2661             {0x1f628, 0x1f62b, 1},
2662             {0x1f62d, 0x1f630, 3},
2663             {0x1f631, 0x1f633, 1},
2664             {0x1f635, 0x1f640, 1},
2665             {0x1f645, 0x1f64f, 1},
2666             {0x1f680, 0x1f6c5, 1},
2667             {0x1f700, 0x1f773, 1},
2668         },
2669     }
2670
2671     var _Z = &RangeTable{
2672         R16: []Range16{
2673             {0x0020, 0x00a0, 128},
2674             {0x1680, 0x180e, 398},
2675             {0x2000, 0x200a, 1},
2676             {0x2028, 0x2029, 1},
2677             {0x202f, 0x205f, 48},
2678             {0x3000, 0x3000, 1},
2679         },
2680     }
2681
2682     var _Zl = &RangeTable{
2683         R16: []Range16{
2684             {0x2028, 0x2028, 1},
2685         },
2686     }
2687
2688     var _Zp = &RangeTable{
2689         R16: []Range16{
2690             {0x2029, 0x2029, 1},
2691         },
2692     }
2693
2694     var _Zs = &RangeTable{
2695         R16: []Range16{
2696             {0x0020, 0x00a0, 128},
2697             {0x1680, 0x180e, 398},
2698             {0x2000, 0x200a, 1},
2699             {0x202f, 0x205f, 48},
2700             {0x3000, 0x3000, 1},
2701         },
2702     }
2703
2704     // The following variables are of type *RangeTable:
2705     var (
2706         Cc      = _Cc // Cc is the set of Unicode characters
2707         Cf      = _Cf // Cf is the set of Unicode characters
2708         Co      = _Co // Co is the set of Unicode characters

```

```

2709     Cs      = _Cs // Cs is the set of Unicode characters
2710     Digit   = _Nd // Digit is the set of Unicode characte
2711     Nd      = _Nd // Nd is the set of Unicode characters
2712     Letter  = _L  // Letter/L is the set of Unicode lette
2713     L       = _L
2714     Lm      = _Lm // Lm is the set of Unicode characters
2715     Lo      = _Lo // Lo is the set of Unicode characters
2716     Lower   = _Ll // Lower is the set of Unicode lower ca
2717     Ll      = _Ll // Ll is the set of Unicode characters
2718     Mark    = _M  // Mark/M is the set of Unicode mark ch
2719     M       = _M
2720     Mc      = _Mc // Mc is the set of Unicode characters
2721     Me      = _Me // Me is the set of Unicode characters
2722     Mn      = _Mn // Mn is the set of Unicode characters
2723     Nl      = _Nl // Nl is the set of Unicode characters
2724     No      = _No // No is the set of Unicode characters
2725     Number  = _N  // Number/N is the set of Unicode numbe
2726     N       = _N
2727     Other   = _C  // Other/C is the set of Unicode control
2728     C       = _C
2729     Pc      = _Pc // Pc is the set of Unicode characters
2730     Pd      = _Pd // Pd is the set of Unicode characters
2731     Pe      = _Pe // Pe is the set of Unicode characters
2732     Pf      = _Pf // Pf is the set of Unicode characters
2733     Pi      = _Pi // Pi is the set of Unicode characters
2734     Po      = _Po // Po is the set of Unicode characters
2735     Ps      = _Ps // Ps is the set of Unicode characters
2736     Punct   = _P  // Punct/P is the set of Unicode punctu
2737     P       = _P
2738     Sc      = _Sc // Sc is the set of Unicode characters
2739     Sk      = _Sk // Sk is the set of Unicode characters
2740     Sm      = _Sm // Sm is the set of Unicode characters
2741     So      = _So // So is the set of Unicode characters
2742     Space   = _Z  // Space/Z is the set of Unicode space
2743     Z       = _Z
2744     Symbol  = _S  // Symbol/S is the set of Unicode symbol
2745     S       = _S
2746     Title   = _Lt // Title is the set of Unicode title ca
2747     Lt      = _Lt // Lt is the set of Unicode characters
2748     Upper   = _Lu // Upper is the set of Unicode upper ca
2749     Lu      = _Lu // Lu is the set of Unicode characters
2750     Zl      = _Zl // Zl is the set of Unicode characters
2751     Zp      = _Zp // Zp is the set of Unicode characters
2752     Zs      = _Zs // Zs is the set of Unicode characters
2753 )
2754
2755 // Generated by running
2756 //      maketables --scripts=all --url=http://www.unicode.or
2757 // DO NOT EDIT
2758

```

```

2759 // Scripts is the set of Unicode script tables.
2760 var Scripts = map[string]*RangeTable{
2761     "Arabic": Arabic,
2762     "Armenian": Armenian,
2763     "Avestan": Avestan,
2764     "Balinese": Balinese,
2765     "Bamum": Bamum,
2766     "Batak": Batak,
2767     "Bengali": Bengali,
2768     "Bopomofo": Bopomofo,
2769     "Brahmi": Brahmi,
2770     "Braille": Braille,
2771     "Buginese": Buginese,
2772     "Buhid": Buhid,
2773     "Canadian_Aboriginal": Canadian_Aboriginal,
2774     "Carian": Carian,
2775     "Cham": Cham,
2776     "Cherokee": Cherokee,
2777     "Common": Common,
2778     "Coptic": Coptic,
2779     "Cuneiform": Cuneiform,
2780     "Cypriot": Cypriot,
2781     "Cyrillic": Cyrillic,
2782     "Deseret": Deseret,
2783     "Devanagari": Devanagari,
2784     "Egyptian_Hieroglyphs": Egyptian_Hieroglyphs,
2785     "Ethiopic": Ethiopic,
2786     "Georgian": Georgian,
2787     "Glagolitic": Glagolitic,
2788     "Gothic": Gothic,
2789     "Greek": Greek,
2790     "Gujarati": Gujarati,
2791     "Gurmukhi": Gurmukhi,
2792     "Han": Han,
2793     "Hangul": Hangul,
2794     "Hanunoo": Hanunoo,
2795     "Hebrew": Hebrew,
2796     "Hiragana": Hiragana,
2797     "Imperial_Aramaic": Imperial_Aramaic,
2798     "Inherited": Inherited,
2799     "Inscriptional_Pahlavi": Inscriptional_Pahlavi,
2800     "Inscriptional_Parthian": Inscriptional_Parthian,
2801     "Javanese": Javanese,
2802     "Kaithi": Kaithi,
2803     "Kannada": Kannada,
2804     "Katakana": Katakana,
2805     "Kayah_Li": Kayah_Li,
2806     "Kharoshthi": Kharoshthi,
2807     "Khmer": Khmer,

```

2808	"Lao":	Lao,
2809	"Latin":	Latin,
2810	"Lepcha":	Lepcha,
2811	"Limbu":	Limbu,
2812	"Linear_B":	Linear_B,
2813	"Lisu":	Lisu,
2814	"Lycian":	Lycian,
2815	"Lydian":	Lydian,
2816	"Malayalam":	Malayalam,
2817	"Mandaic":	Mandaic,
2818	"Meetei_Mayek":	Meetei_Mayek,
2819	"Mongolian":	Mongolian,
2820	"Myanmar":	Myanmar,
2821	"New_Tai_Lue":	New_Tai_Lue,
2822	"Nko":	Nko,
2823	"Ogham":	Ogham,
2824	"Ol_Chiki":	Ol_Chiki,
2825	"Old_Italic":	Old_Italic,
2826	"Old_Persian":	Old_Persian,
2827	"Old_South_Arabian":	Old_South_Arabian,
2828	"Old_Turkic":	Old_Turkic,
2829	"Oriya":	Oriya,
2830	"Osmanya":	Osmanya,
2831	"Phags_Pa":	Phags_Pa,
2832	"Phoenician":	Phoenician,
2833	"Rejang":	Rejang,
2834	"Runic":	Runic,
2835	"Samaritan":	Samaritan,
2836	"Saurashtra":	Saurashtra,
2837	"Shavian":	Shavian,
2838	"Sinhala":	Sinhala,
2839	"Sundanese":	Sundanese,
2840	"Syloti_Nagri":	Syloti_Nagri,
2841	"Syriac":	Syriac,
2842	"Tagalog":	Tagalog,
2843	"Tagbanwa":	Tagbanwa,
2844	"Tai_Le":	Tai_Le,
2845	"Tai_Tham":	Tai_Tham,
2846	"Tai_Viet":	Tai_Viet,
2847	"Tamil":	Tamil,
2848	"Telugu":	Telugu,
2849	"Thaana":	Thaana,
2850	"Thai":	Thai,
2851	"Tibetan":	Tibetan,
2852	"Tifinagh":	Tifinagh,
2853	"Ugaritic":	Ugaritic,
2854	"Vai":	Vai,
2855	"Yi":	Yi,
2856	}	

```

2857
2858 var _Arabic = &RangeTable{
2859     R16: []Range16{
2860         {0x0600, 0x0603, 1},
2861         {0x0606, 0x060b, 1},
2862         {0x060d, 0x061a, 1},
2863         {0x061e, 0x061e, 1},
2864         {0x0620, 0x063f, 1},
2865         {0x0641, 0x064a, 1},
2866         {0x0656, 0x065e, 1},
2867         {0x066a, 0x066f, 1},
2868         {0x0671, 0x06dc, 1},
2869         {0x06de, 0x06ff, 1},
2870         {0x0750, 0x077f, 1},
2871         {0xfb50, 0xfbc1, 1},
2872         {0xfbdb, 0xfd3d, 1},
2873         {0xfd50, 0xfd8f, 1},
2874         {0xfd92, 0xfdc7, 1},
2875         {0xfdf0, 0xfdfc, 1},
2876         {0xfe70, 0xfe74, 1},
2877         {0xfe76, 0xfefc, 1},
2878     },
2879     R32: []Range32{
2880         {0x10e60, 0x10e7e, 1},
2881     },
2882 }
2883
2884 var _Armenian = &RangeTable{
2885     R16: []Range16{
2886         {0x0531, 0x0556, 1},
2887         {0x0559, 0x055f, 1},
2888         {0x0561, 0x0587, 1},
2889         {0x058a, 0x058a, 1},
2890         {0xfb13, 0xfb17, 1},
2891     },
2892 }
2893
2894 var _Avestan = &RangeTable{
2895     R16: []Range16{},
2896     R32: []Range32{
2897         {0x10b00, 0x10b35, 1},
2898         {0x10b39, 0x10b3f, 1},
2899     },
2900 }
2901
2902 var _Balinese = &RangeTable{
2903     R16: []Range16{
2904         {0x1b00, 0x1b4b, 1},
2905         {0x1b50, 0x1b7c, 1},
2906     },

```

```

2907 }
2908
2909 var _Bamum = &RangeTable{
2910     R16: []Range16{
2911         {0xa6a0, 0xa6f7, 1},
2912     },
2913     R32: []Range32{
2914         {0x16800, 0x16a38, 1},
2915     },
2916 }
2917
2918 var _Batak = &RangeTable{
2919     R16: []Range16{
2920         {0x1bc0, 0x1bf3, 1},
2921         {0x1bfc, 0x1bff, 1},
2922     },
2923 }
2924
2925 var _Bengali = &RangeTable{
2926     R16: []Range16{
2927         {0x0981, 0x0983, 1},
2928         {0x0985, 0x098c, 1},
2929         {0x098f, 0x0990, 1},
2930         {0x0993, 0x09a8, 1},
2931         {0x09aa, 0x09b0, 1},
2932         {0x09b2, 0x09b2, 1},
2933         {0x09b6, 0x09b9, 1},
2934         {0x09bc, 0x09c4, 1},
2935         {0x09c7, 0x09c8, 1},
2936         {0x09cb, 0x09ce, 1},
2937         {0x09d7, 0x09d7, 1},
2938         {0x09dc, 0x09dd, 1},
2939         {0x09df, 0x09e3, 1},
2940         {0x09e6, 0x09fb, 1},
2941     },
2942 }
2943
2944 var _Bopomofo = &RangeTable{
2945     R16: []Range16{
2946         {0x02ea, 0x02eb, 1},
2947         {0x3105, 0x312d, 1},
2948         {0x31a0, 0x31ba, 1},
2949     },
2950 }
2951
2952 var _Brahmi = &RangeTable{
2953     R16: []Range16{},
2954     R32: []Range32{
2955         {0x11000, 0x1104d, 1},

```

```

2956             {0x11052, 0x1106f, 1},
2957         },
2958     }
2959
2960     var _Braille = &RangeTable{
2961         R16: []Range16{
2962             {0x2800, 0x28ff, 1},
2963         },
2964     }
2965
2966     var _Buginese = &RangeTable{
2967         R16: []Range16{
2968             {0x1a00, 0x1a1b, 1},
2969             {0x1a1e, 0x1a1f, 1},
2970         },
2971     }
2972
2973     var _Buhid = &RangeTable{
2974         R16: []Range16{
2975             {0x1740, 0x1753, 1},
2976         },
2977     }
2978
2979     var _Canadian_Aboriginal = &RangeTable{
2980         R16: []Range16{
2981             {0x1400, 0x167f, 1},
2982             {0x18b0, 0x18f5, 1},
2983         },
2984     }
2985
2986     var _Carian = &RangeTable{
2987         R16: []Range16{},
2988         R32: []Range32{
2989             {0x102a0, 0x102d0, 1},
2990         },
2991     }
2992
2993     var _Cham = &RangeTable{
2994         R16: []Range16{
2995             {0xaa00, 0xaa36, 1},
2996             {0xaa40, 0xaa4d, 1},
2997             {0xaa50, 0xaa59, 1},
2998             {0xaa5c, 0xaa5f, 1},
2999         },
3000     }
3001
3002     var _Cherokee = &RangeTable{
3003         R16: []Range16{
3004             {0x13a0, 0x13f4, 1},

```

```
3005     },
3006 }
3007
3008 var _Common = &RangeTable{
3009     R16: []Range16{
3010         {0x0000, 0x0040, 1},
3011         {0x005b, 0x0060, 1},
3012         {0x007b, 0x00a9, 1},
3013         {0x00ab, 0x00b9, 1},
3014         {0x00bb, 0x00bf, 1},
3015         {0x00d7, 0x00d7, 1},
3016         {0x00f7, 0x00f7, 1},
3017         {0x02b9, 0x02df, 1},
3018         {0x02e5, 0x02e9, 1},
3019         {0x02ec, 0x02ff, 1},
3020         {0x0374, 0x0374, 1},
3021         {0x037e, 0x037e, 1},
3022         {0x0385, 0x0385, 1},
3023         {0x0387, 0x0387, 1},
3024         {0x0589, 0x0589, 1},
3025         {0x060c, 0x060c, 1},
3026         {0x061b, 0x061b, 1},
3027         {0x061f, 0x061f, 1},
3028         {0x0640, 0x0640, 1},
3029         {0x0660, 0x0669, 1},
3030         {0x06dd, 0x06dd, 1},
3031         {0x0964, 0x0965, 1},
3032         {0x0970, 0x0970, 1},
3033         {0x0e3f, 0x0e3f, 1},
3034         {0x0fd5, 0x0fd8, 1},
3035         {0x10fb, 0x10fb, 1},
3036         {0x16eb, 0x16ed, 1},
3037         {0x1735, 0x1736, 1},
3038         {0x1802, 0x1803, 1},
3039         {0x1805, 0x1805, 1},
3040         {0x1cd3, 0x1cd3, 1},
3041         {0x1ce1, 0x1ce1, 1},
3042         {0x1ce9, 0x1cec, 1},
3043         {0x1cee, 0x1cf2, 1},
3044         {0x2000, 0x200b, 1},
3045         {0x200e, 0x2064, 1},
3046         {0x206a, 0x2070, 1},
3047         {0x2074, 0x207e, 1},
3048         {0x2080, 0x208e, 1},
3049         {0x20a0, 0x20b9, 1},
3050         {0x2100, 0x2125, 1},
3051         {0x2127, 0x2129, 1},
3052         {0x212c, 0x2131, 1},
3053         {0x2133, 0x214d, 1},
3054         {0x214f, 0x215f, 1},
```

```

3055      {0x2189, 0x2189, 1},
3056      {0x2190, 0x23f3, 1},
3057      {0x2400, 0x2426, 1},
3058      {0x2440, 0x244a, 1},
3059      {0x2460, 0x26ff, 1},
3060      {0x2701, 0x27ca, 1},
3061      {0x27cc, 0x27cc, 1},
3062      {0x27ce, 0x27ff, 1},
3063      {0x2900, 0x2b4c, 1},
3064      {0x2b50, 0x2b59, 1},
3065      {0x2e00, 0x2e31, 1},
3066      {0x2ff0, 0x2ffb, 1},
3067      {0x3000, 0x3004, 1},
3068      {0x3006, 0x3006, 1},
3069      {0x3008, 0x3020, 1},
3070      {0x3030, 0x3037, 1},
3071      {0x303c, 0x303f, 1},
3072      {0x309b, 0x309c, 1},
3073      {0x30a0, 0x30a0, 1},
3074      {0x30fb, 0x30fc, 1},
3075      {0x3190, 0x319f, 1},
3076      {0x31c0, 0x31e3, 1},
3077      {0x3220, 0x325f, 1},
3078      {0x327f, 0x32cf, 1},
3079      {0x3358, 0x33ff, 1},
3080      {0x4dc0, 0x4dff, 1},
3081      {0xa700, 0xa721, 1},
3082      {0xa788, 0xa78a, 1},
3083      {0xa830, 0xa839, 1},
3084      {0xfd3e, 0xfd3f, 1},
3085      {0xfdfd, 0xfdfd, 1},
3086      {0xfe10, 0xfe19, 1},
3087      {0xfe30, 0xfe52, 1},
3088      {0xfe54, 0xfe66, 1},
3089      {0xfe68, 0xfe6b, 1},
3090      {0xfeff, 0xfeff, 1},
3091      {0xff01, 0xff20, 1},
3092      {0xff3b, 0xff40, 1},
3093      {0xff5b, 0xff65, 1},
3094      {0xff70, 0xff70, 1},
3095      {0xff9e, 0xff9f, 1},
3096      {0xffe0, 0xffe6, 1},
3097      {0xffe8, 0xffee, 1},
3098      {0xffff9, 0xffffd, 1},
3099      },
3100      R32: []Range32{
3101          {0x10100, 0x10102, 1},
3102          {0x10107, 0x10133, 1},
3103          {0x10137, 0x1013f, 1},

```

3104	{0x10190, 0x1019b, 1},
3105	{0x101d0, 0x101fc, 1},
3106	{0x1d000, 0x1d0f5, 1},
3107	{0x1d100, 0x1d126, 1},
3108	{0x1d129, 0x1d166, 1},
3109	{0x1d16a, 0x1d17a, 1},
3110	{0x1d183, 0x1d184, 1},
3111	{0x1d18c, 0x1d1a9, 1},
3112	{0x1d1ae, 0x1d1dd, 1},
3113	{0x1d300, 0x1d356, 1},
3114	{0x1d360, 0x1d371, 1},
3115	{0x1d400, 0x1d454, 1},
3116	{0x1d456, 0x1d49c, 1},
3117	{0x1d49e, 0x1d49f, 1},
3118	{0x1d4a2, 0x1d4a2, 1},
3119	{0x1d4a5, 0x1d4a6, 1},
3120	{0x1d4a9, 0x1d4ac, 1},
3121	{0x1d4ae, 0x1d4b9, 1},
3122	{0x1d4bb, 0x1d4bb, 1},
3123	{0x1d4bd, 0x1d4c3, 1},
3124	{0x1d4c5, 0x1d505, 1},
3125	{0x1d507, 0x1d50a, 1},
3126	{0x1d50d, 0x1d514, 1},
3127	{0x1d516, 0x1d51c, 1},
3128	{0x1d51e, 0x1d539, 1},
3129	{0x1d53b, 0x1d53e, 1},
3130	{0x1d540, 0x1d544, 1},
3131	{0x1d546, 0x1d546, 1},
3132	{0x1d54a, 0x1d550, 1},
3133	{0x1d552, 0x1d6a5, 1},
3134	{0x1d6a8, 0x1d7cb, 1},
3135	{0x1d7ce, 0x1d7ff, 1},
3136	{0x1f000, 0x1f02b, 1},
3137	{0x1f030, 0x1f093, 1},
3138	{0x1f0a0, 0x1f0ae, 1},
3139	{0x1f0b1, 0x1f0be, 1},
3140	{0x1f0c1, 0x1f0cf, 1},
3141	{0x1f0d1, 0x1f0df, 1},
3142	{0x1f100, 0x1f10a, 1},
3143	{0x1f110, 0x1f12e, 1},
3144	{0x1f130, 0x1f169, 1},
3145	{0x1f170, 0x1f19a, 1},
3146	{0x1f1e6, 0x1f1ff, 1},
3147	{0x1f201, 0x1f202, 1},
3148	{0x1f210, 0x1f23a, 1},
3149	{0x1f240, 0x1f248, 1},
3150	{0x1f250, 0x1f251, 1},
3151	{0x1f300, 0x1f320, 1},
3152	{0x1f330, 0x1f335, 1},

```

3153         {0x1f337, 0x1f37c, 1},
3154         {0x1f380, 0x1f393, 1},
3155         {0x1f3a0, 0x1f3c4, 1},
3156         {0x1f3c6, 0x1f3ca, 1},
3157         {0x1f3e0, 0x1f3f0, 1},
3158         {0x1f400, 0x1f43e, 1},
3159         {0x1f440, 0x1f440, 1},
3160         {0x1f442, 0x1f4f7, 1},
3161         {0x1f4f9, 0x1f4fc, 1},
3162         {0x1f500, 0x1f53d, 1},
3163         {0x1f550, 0x1f567, 1},
3164         {0x1f5fb, 0x1f5ff, 1},
3165         {0x1f601, 0x1f610, 1},
3166         {0x1f612, 0x1f614, 1},
3167         {0x1f616, 0x1f616, 1},
3168         {0x1f618, 0x1f618, 1},
3169         {0x1f61a, 0x1f61a, 1},
3170         {0x1f61c, 0x1f61e, 1},
3171         {0x1f620, 0x1f625, 1},
3172         {0x1f628, 0x1f62b, 1},
3173         {0x1f62d, 0x1f62d, 1},
3174         {0x1f630, 0x1f633, 1},
3175         {0x1f635, 0x1f640, 1},
3176         {0x1f645, 0x1f64f, 1},
3177         {0x1f680, 0x1f6c5, 1},
3178         {0x1f700, 0x1f773, 1},
3179         {0xe0001, 0xe0001, 1},
3180         {0xe0020, 0xe007f, 1},
3181     },
3182 }
3183
3184 var _Coptic = &RangeTable{
3185     R16: []Range16{
3186         {0x03e2, 0x03ef, 1},
3187         {0x2c80, 0x2cf1, 1},
3188         {0x2cf9, 0x2cff, 1},
3189     },
3190 }
3191
3192 var _Cuneiform = &RangeTable{
3193     R16: []Range16{},
3194     R32: []Range32{
3195         {0x12000, 0x1236e, 1},
3196         {0x12400, 0x12462, 1},
3197         {0x12470, 0x12473, 1},
3198     },
3199 }
3200
3201 var _Cypriot = &RangeTable{
3202     R16: []Range16{},

```

```

3203         R32: []Range32{
3204             {0x10800, 0x10805, 1},
3205             {0x10808, 0x10808, 1},
3206             {0x1080a, 0x10835, 1},
3207             {0x10837, 0x10838, 1},
3208             {0x1083c, 0x1083c, 1},
3209             {0x1083f, 0x1083f, 1},
3210         },
3211     }
3212
3213     var _Cyrillic = &RangeTable{
3214         R16: []Range16{
3215             {0x0400, 0x0484, 1},
3216             {0x0487, 0x0527, 1},
3217             {0x1d2b, 0x1d2b, 1},
3218             {0x1d78, 0x1d78, 1},
3219             {0x2de0, 0x2dff, 1},
3220             {0xa640, 0xa673, 1},
3221             {0xa67c, 0xa697, 1},
3222         },
3223     }
3224
3225     var _Deseret = &RangeTable{
3226         R16: []Range16{},
3227         R32: []Range32{
3228             {0x10400, 0x1044f, 1},
3229         },
3230     }
3231
3232     var _Devanagari = &RangeTable{
3233         R16: []Range16{
3234             {0x0900, 0x0950, 1},
3235             {0x0953, 0x0963, 1},
3236             {0x0966, 0x096f, 1},
3237             {0x0971, 0x0977, 1},
3238             {0x0979, 0x097f, 1},
3239             {0xa8e0, 0xa8fb, 1},
3240         },
3241     }
3242
3243     var _Egyptian_Hieroglyphs = &RangeTable{
3244         R16: []Range16{},
3245         R32: []Range32{
3246             {0x13000, 0x1342e, 1},
3247         },
3248     }
3249
3250     var _Ethiopic = &RangeTable{
3251         R16: []Range16{

```

```

3252         {0x1200, 0x1248, 1},
3253         {0x124a, 0x124d, 1},
3254         {0x1250, 0x1256, 1},
3255         {0x1258, 0x1258, 1},
3256         {0x125a, 0x125d, 1},
3257         {0x1260, 0x1288, 1},
3258         {0x128a, 0x128d, 1},
3259         {0x1290, 0x12b0, 1},
3260         {0x12b2, 0x12b5, 1},
3261         {0x12b8, 0x12be, 1},
3262         {0x12c0, 0x12c0, 1},
3263         {0x12c2, 0x12c5, 1},
3264         {0x12c8, 0x12d6, 1},
3265         {0x12d8, 0x1310, 1},
3266         {0x1312, 0x1315, 1},
3267         {0x1318, 0x135a, 1},
3268         {0x135d, 0x137c, 1},
3269         {0x1380, 0x1399, 1},
3270         {0x2d80, 0x2d96, 1},
3271         {0x2da0, 0x2da6, 1},
3272         {0x2da8, 0x2dae, 1},
3273         {0x2db0, 0x2db6, 1},
3274         {0x2db8, 0x2dbe, 1},
3275         {0x2dc0, 0x2dc6, 1},
3276         {0x2dc8, 0x2dce, 1},
3277         {0x2dd0, 0x2dd6, 1},
3278         {0x2dd8, 0x2dde, 1},
3279         {0xab01, 0xab06, 1},
3280         {0xab09, 0xab0e, 1},
3281         {0xab11, 0xab16, 1},
3282         {0xab20, 0xab26, 1},
3283         {0xab28, 0xab2e, 1},
3284     },
3285 }
3286
3287 var _Georgian = &RangeTable{
3288     R16: []Range16{
3289         {0x10a0, 0x10c5, 1},
3290         {0x10d0, 0x10fa, 1},
3291         {0x10fc, 0x10fc, 1},
3292         {0x2d00, 0x2d25, 1},
3293     },
3294 }
3295
3296 var _Glagolitic = &RangeTable{
3297     R16: []Range16{
3298         {0x2c00, 0x2c2e, 1},
3299         {0x2c30, 0x2c5e, 1},
3300     },

```

```

3301 }
3302
3303 var _Gothic = &RangeTable{
3304     R16: []Range16{},
3305     R32: []Range32{
3306         {0x10330, 0x1034a, 1},
3307     },
3308 }
3309
3310 var _Greek = &RangeTable{
3311     R16: []Range16{
3312         {0x0370, 0x0373, 1},
3313         {0x0375, 0x0377, 1},
3314         {0x037a, 0x037d, 1},
3315         {0x0384, 0x0384, 1},
3316         {0x0386, 0x0386, 1},
3317         {0x0388, 0x038a, 1},
3318         {0x038c, 0x038c, 1},
3319         {0x038e, 0x03a1, 1},
3320         {0x03a3, 0x03e1, 1},
3321         {0x03f0, 0x03ff, 1},
3322         {0x1d26, 0x1d2a, 1},
3323         {0x1d5d, 0x1d61, 1},
3324         {0x1d66, 0x1d6a, 1},
3325         {0x1dbf, 0x1dbf, 1},
3326         {0x1f00, 0x1f15, 1},
3327         {0x1f18, 0x1f1d, 1},
3328         {0x1f20, 0x1f45, 1},
3329         {0x1f48, 0x1f4d, 1},
3330         {0x1f50, 0x1f57, 1},
3331         {0x1f59, 0x1f59, 1},
3332         {0x1f5b, 0x1f5b, 1},
3333         {0x1f5d, 0x1f5d, 1},
3334         {0x1f5f, 0x1f7d, 1},
3335         {0x1f80, 0x1fb4, 1},
3336         {0x1fb6, 0x1fc4, 1},
3337         {0x1fc6, 0x1fd3, 1},
3338         {0x1fd6, 0x1fdb, 1},
3339         {0x1fdd, 0x1fef, 1},
3340         {0x1ff2, 0x1ff4, 1},
3341         {0x1ff6, 0x1ffe, 1},
3342         {0x2126, 0x2126, 1},
3343     },
3344     R32: []Range32{
3345         {0x10140, 0x1018a, 1},
3346         {0x1d200, 0x1d245, 1},
3347     },
3348 }
3349
3350 var _Gujarati = &RangeTable{

```

```

3351         R16: []Range16{
3352             {0x0a81, 0x0a83, 1},
3353             {0x0a85, 0x0a8d, 1},
3354             {0x0a8f, 0x0a91, 1},
3355             {0x0a93, 0x0aa8, 1},
3356             {0x0aaa, 0x0ab0, 1},
3357             {0x0ab2, 0x0ab3, 1},
3358             {0x0ab5, 0x0ab9, 1},
3359             {0x0abc, 0x0ac5, 1},
3360             {0x0ac7, 0x0ac9, 1},
3361             {0x0acb, 0x0acd, 1},
3362             {0x0ad0, 0x0ad0, 1},
3363             {0x0ae0, 0x0ae3, 1},
3364             {0x0ae6, 0x0aef, 1},
3365             {0x0af1, 0x0af1, 1},
3366         },
3367     }
3368
3369     var _Gurmukhi = &RangeTable{
3370         R16: []Range16{
3371             {0x0a01, 0x0a03, 1},
3372             {0x0a05, 0x0a0a, 1},
3373             {0x0a0f, 0x0a10, 1},
3374             {0x0a13, 0x0a28, 1},
3375             {0x0a2a, 0x0a30, 1},
3376             {0x0a32, 0x0a33, 1},
3377             {0x0a35, 0x0a36, 1},
3378             {0x0a38, 0x0a39, 1},
3379             {0x0a3c, 0x0a3c, 1},
3380             {0x0a3e, 0x0a42, 1},
3381             {0x0a47, 0x0a48, 1},
3382             {0x0a4b, 0x0a4d, 1},
3383             {0x0a51, 0x0a51, 1},
3384             {0x0a59, 0x0a5c, 1},
3385             {0x0a5e, 0x0a5e, 1},
3386             {0x0a66, 0x0a75, 1},
3387         },
3388     }
3389
3390     var _Han = &RangeTable{
3391         R16: []Range16{
3392             {0x2e80, 0x2e99, 1},
3393             {0x2e9b, 0x2ef3, 1},
3394             {0x2f00, 0x2fd5, 1},
3395             {0x3005, 0x3005, 1},
3396             {0x3007, 0x3007, 1},
3397             {0x3021, 0x3029, 1},
3398             {0x3038, 0x303b, 1},
3399             {0x3400, 0x4db5, 1},

```

```

3400             {0x4e00, 0x9fcb, 1},
3401             {0xf900, 0xfa2d, 1},
3402             {0xfa30, 0xfa6d, 1},
3403             {0xfa70, 0xfad9, 1},
3404         },
3405         R32: []Range32{
3406             {0x20000, 0x2a6d6, 1},
3407             {0x2a700, 0x2b734, 1},
3408             {0x2b740, 0x2b81d, 1},
3409             {0x2f800, 0x2fa1d, 1},
3410         },
3411     }
3412
3413     var _Hangul = &RangeTable{
3414         R16: []Range16{
3415             {0x1100, 0x11ff, 1},
3416             {0x302e, 0x302f, 1},
3417             {0x3131, 0x318e, 1},
3418             {0x3200, 0x321e, 1},
3419             {0x3260, 0x327e, 1},
3420             {0xa960, 0xa97c, 1},
3421             {0xac00, 0xd7a3, 1},
3422             {0xd7b0, 0xd7c6, 1},
3423             {0xd7cb, 0xd7fb, 1},
3424             {0xffa0, 0xffbe, 1},
3425             {0xffc2, 0xffc7, 1},
3426             {0xffca, 0xffcf, 1},
3427             {0xffd2, 0xffd7, 1},
3428             {0xffda, 0xffdc, 1},
3429         },
3430     }
3431
3432     var _Hanunoo = &RangeTable{
3433         R16: []Range16{
3434             {0x1720, 0x1734, 1},
3435         },
3436     }
3437
3438     var _Hebrew = &RangeTable{
3439         R16: []Range16{
3440             {0x0591, 0x05c7, 1},
3441             {0x05d0, 0x05ea, 1},
3442             {0x05f0, 0x05f4, 1},
3443             {0xfb1d, 0xfb36, 1},
3444             {0xfb38, 0xfb3c, 1},
3445             {0xfb3e, 0xfb3e, 1},
3446             {0xfb40, 0xfb41, 1},
3447             {0xfb43, 0xfb44, 1},
3448             {0xfb46, 0xfb4f, 1},

```

```

3449     },
3450 }
3451
3452 var _Hiragana = &RangeTable{
3453     R16: []Range16{
3454         {0x3041, 0x3096, 1},
3455         {0x309d, 0x309f, 1},
3456     },
3457     R32: []Range32{
3458         {0x1b001, 0x1b001, 1},
3459         {0x1f200, 0x1f200, 1},
3460     },
3461 }
3462
3463 var _Imperial_Aramaic = &RangeTable{
3464     R16: []Range16{},
3465     R32: []Range32{
3466         {0x10840, 0x10855, 1},
3467         {0x10857, 0x1085f, 1},
3468     },
3469 }
3470
3471 var _Inherited = &RangeTable{
3472     R16: []Range16{
3473         {0x0300, 0x036f, 1},
3474         {0x0485, 0x0486, 1},
3475         {0x064b, 0x0655, 1},
3476         {0x065f, 0x065f, 1},
3477         {0x0670, 0x0670, 1},
3478         {0x0951, 0x0952, 1},
3479         {0x1cd0, 0x1cd2, 1},
3480         {0x1cd4, 0x1ce0, 1},
3481         {0x1ce2, 0x1ce8, 1},
3482         {0x1ced, 0x1ced, 1},
3483         {0x1dc0, 0x1de6, 1},
3484         {0x1dfc, 0x1dff, 1},
3485         {0x200c, 0x200d, 1},
3486         {0x20d0, 0x20f0, 1},
3487         {0x302a, 0x302d, 1},
3488         {0x3099, 0x309a, 1},
3489         {0xfe00, 0xfe0f, 1},
3490         {0xfe20, 0xfe26, 1},
3491     },
3492     R32: []Range32{
3493         {0x101fd, 0x101fd, 1},
3494         {0x1d167, 0x1d169, 1},
3495         {0x1d17b, 0x1d182, 1},
3496         {0x1d185, 0x1d18b, 1},
3497         {0x1d1aa, 0x1d1ad, 1},
3498         {0xe0100, 0xe01ef, 1},

```

```

3499     },
3500 }
3501
3502 var _Inscriptional_Pahlavi = &RangeTable{
3503     R16: []Range16{},
3504     R32: []Range32{
3505         {0x10b60, 0x10b72, 1},
3506         {0x10b78, 0x10b7f, 1},
3507     },
3508 }
3509
3510 var _Inscriptional_Parthian = &RangeTable{
3511     R16: []Range16{},
3512     R32: []Range32{
3513         {0x10b40, 0x10b55, 1},
3514         {0x10b58, 0x10b5f, 1},
3515     },
3516 }
3517
3518 var _Javanese = &RangeTable{
3519     R16: []Range16{
3520         {0xa980, 0xa9cd, 1},
3521         {0xa9cf, 0xa9d9, 1},
3522         {0xa9de, 0xa9df, 1},
3523     },
3524 }
3525
3526 var _Kaithi = &RangeTable{
3527     R16: []Range16{},
3528     R32: []Range32{
3529         {0x11080, 0x110c1, 1},
3530     },
3531 }
3532
3533 var _Kannada = &RangeTable{
3534     R16: []Range16{
3535         {0x0c82, 0x0c83, 1},
3536         {0x0c85, 0x0c8c, 1},
3537         {0x0c8e, 0x0c90, 1},
3538         {0x0c92, 0x0ca8, 1},
3539         {0x0caa, 0x0cb3, 1},
3540         {0x0cb5, 0x0cb9, 1},
3541         {0x0cbc, 0x0cc4, 1},
3542         {0x0cc6, 0x0cc8, 1},
3543         {0x0cca, 0x0ccd, 1},
3544         {0x0cd5, 0x0cd6, 1},
3545         {0x0cde, 0x0cde, 1},
3546         {0x0ce0, 0x0ce3, 1},
3547         {0x0ce6, 0x0cef, 1},

```

```

3548             {0x0cf1, 0x0cf2, 1},
3549         },
3550     }
3551
3552     var _Katakana = &RangeTable{
3553         R16: []Range16{
3554             {0x30a1, 0x30fa, 1},
3555             {0x30fd, 0x30ff, 1},
3556             {0x31f0, 0x31ff, 1},
3557             {0x32d0, 0x32fe, 1},
3558             {0x3300, 0x3357, 1},
3559             {0xff66, 0xff6f, 1},
3560             {0xff71, 0xff9d, 1},
3561         },
3562         R32: []Range32{
3563             {0x1b000, 0x1b000, 1},
3564         },
3565     }
3566
3567     var _Kayah_Li = &RangeTable{
3568         R16: []Range16{
3569             {0xa900, 0xa92f, 1},
3570         },
3571     }
3572
3573     var _Kharoshthi = &RangeTable{
3574         R16: []Range16{},
3575         R32: []Range32{
3576             {0x10a00, 0x10a03, 1},
3577             {0x10a05, 0x10a06, 1},
3578             {0x10a0c, 0x10a13, 1},
3579             {0x10a15, 0x10a17, 1},
3580             {0x10a19, 0x10a33, 1},
3581             {0x10a38, 0x10a3a, 1},
3582             {0x10a3f, 0x10a47, 1},
3583             {0x10a50, 0x10a58, 1},
3584         },
3585     }
3586
3587     var _Khmer = &RangeTable{
3588         R16: []Range16{
3589             {0x1780, 0x17dd, 1},
3590             {0x17e0, 0x17e9, 1},
3591             {0x17f0, 0x17f9, 1},
3592             {0x19e0, 0x19ff, 1},
3593         },
3594     }
3595
3596     var _Lao = &RangeTable{

```

```

3597         R16: []Range16{
3598             {0x0e81, 0x0e82, 1},
3599             {0x0e84, 0x0e84, 1},
3600             {0x0e87, 0x0e88, 1},
3601             {0x0e8a, 0x0e8a, 1},
3602             {0x0e8d, 0x0e8d, 1},
3603             {0x0e94, 0x0e97, 1},
3604             {0x0e99, 0x0e9f, 1},
3605             {0x0ea1, 0x0ea3, 1},
3606             {0x0ea5, 0x0ea5, 1},
3607             {0x0ea7, 0x0ea7, 1},
3608             {0x0eaa, 0x0eab, 1},
3609             {0x0ead, 0x0eb9, 1},
3610             {0x0ebb, 0x0ebd, 1},
3611             {0x0ec0, 0x0ec4, 1},
3612             {0x0ec6, 0x0ec6, 1},
3613             {0x0ec8, 0x0ecd, 1},
3614             {0x0ed0, 0x0ed9, 1},
3615             {0x0edc, 0x0edd, 1},
3616         },
3617     }
3618
3619     var _Latin = &RangeTable{
3620         R16: []Range16{
3621             {0x0041, 0x005a, 1},
3622             {0x0061, 0x007a, 1},
3623             {0x00aa, 0x00aa, 1},
3624             {0x00ba, 0x00ba, 1},
3625             {0x00c0, 0x00d6, 1},
3626             {0x00d8, 0x00f6, 1},
3627             {0x00f8, 0x02b8, 1},
3628             {0x02e0, 0x02e4, 1},
3629             {0x1d00, 0x1d25, 1},
3630             {0x1d2c, 0x1d5c, 1},
3631             {0x1d62, 0x1d65, 1},
3632             {0x1d6b, 0x1d77, 1},
3633             {0x1d79, 0x1dbe, 1},
3634             {0x1e00, 0x1eff, 1},
3635             {0x2071, 0x2071, 1},
3636             {0x207f, 0x207f, 1},
3637             {0x2090, 0x209c, 1},
3638             {0x212a, 0x212b, 1},
3639             {0x2132, 0x2132, 1},
3640             {0x214e, 0x214e, 1},
3641             {0x2160, 0x2188, 1},
3642             {0x2c60, 0x2c7f, 1},
3643             {0xa722, 0xa787, 1},
3644             {0xa78b, 0xa78e, 1},
3645             {0xa790, 0xa791, 1},
3646             {0xa7a0, 0xa7a9, 1},

```

```

3647             {0xa7fa, 0xa7ff, 1},
3648             {0xfb00, 0xfb06, 1},
3649             {0xff21, 0xff3a, 1},
3650             {0xff41, 0xff5a, 1},
3651         },
3652     }
3653
3654     var _Lepcha = &RangeTable{
3655         R16: []Range16{
3656             {0x1c00, 0x1c37, 1},
3657             {0x1c3b, 0x1c49, 1},
3658             {0x1c4d, 0x1c4f, 1},
3659         },
3660     }
3661
3662     var _Limbu = &RangeTable{
3663         R16: []Range16{
3664             {0x1900, 0x191c, 1},
3665             {0x1920, 0x192b, 1},
3666             {0x1930, 0x193b, 1},
3667             {0x1940, 0x1940, 1},
3668             {0x1944, 0x194f, 1},
3669         },
3670     }
3671
3672     var _Linear_B = &RangeTable{
3673         R16: []Range16{},
3674         R32: []Range32{
3675             {0x10000, 0x1000b, 1},
3676             {0x1000d, 0x10026, 1},
3677             {0x10028, 0x1003a, 1},
3678             {0x1003c, 0x1003d, 1},
3679             {0x1003f, 0x1004d, 1},
3680             {0x10050, 0x1005d, 1},
3681             {0x10080, 0x100fa, 1},
3682         },
3683     }
3684
3685     var _Lisu = &RangeTable{
3686         R16: []Range16{
3687             {0xa4d0, 0xa4ff, 1},
3688         },
3689     }
3690
3691     var _Lycian = &RangeTable{
3692         R16: []Range16{},
3693         R32: []Range32{
3694             {0x10280, 0x1029c, 1},
3695         },

```

```

3696 }
3697
3698 var _Lydian = &RangeTable{
3699     R16: []Range16{,
3700         R32: []Range32{
3701             {0x10920, 0x10939, 1},
3702             {0x1093f, 0x1093f, 1},
3703         },
3704     }
3705
3706 var _Malayalam = &RangeTable{
3707     R16: []Range16{
3708         {0x0d02, 0x0d03, 1},
3709         {0x0d05, 0x0d0c, 1},
3710         {0x0d0e, 0x0d10, 1},
3711         {0x0d12, 0x0d3a, 1},
3712         {0x0d3d, 0x0d44, 1},
3713         {0x0d46, 0x0d48, 1},
3714         {0x0d4a, 0x0d4e, 1},
3715         {0x0d57, 0x0d57, 1},
3716         {0x0d60, 0x0d63, 1},
3717         {0x0d66, 0x0d75, 1},
3718         {0x0d79, 0x0d7f, 1},
3719     },
3720 }
3721
3722 var _Mandaic = &RangeTable{
3723     R16: []Range16{
3724         {0x0840, 0x085b, 1},
3725         {0x085e, 0x085e, 1},
3726     },
3727 }
3728
3729 var _Meetei_Mayek = &RangeTable{
3730     R16: []Range16{
3731         {0xabc0, 0xabed, 1},
3732         {0xabf0, 0xabf9, 1},
3733     },
3734 }
3735
3736 var _Mongolian = &RangeTable{
3737     R16: []Range16{
3738         {0x1800, 0x1801, 1},
3739         {0x1804, 0x1804, 1},
3740         {0x1806, 0x180e, 1},
3741         {0x1810, 0x1819, 1},
3742         {0x1820, 0x1877, 1},
3743         {0x1880, 0x18aa, 1},
3744     },

```

```

3745 }
3746
3747 var _Myanmar = &RangeTable{
3748     R16: []Range16{
3749         {0x1000, 0x109f, 1},
3750         {0xaa60, 0xaa7b, 1},
3751     },
3752 }
3753
3754 var _New_Tai_Lue = &RangeTable{
3755     R16: []Range16{
3756         {0x1980, 0x19ab, 1},
3757         {0x19b0, 0x19c9, 1},
3758         {0x19d0, 0x19da, 1},
3759         {0x19de, 0x19df, 1},
3760     },
3761 }
3762
3763 var _Nko = &RangeTable{
3764     R16: []Range16{
3765         {0x07c0, 0x07fa, 1},
3766     },
3767 }
3768
3769 var _Ogham = &RangeTable{
3770     R16: []Range16{
3771         {0x1680, 0x169c, 1},
3772     },
3773 }
3774
3775 var _Ol_Chiki = &RangeTable{
3776     R16: []Range16{
3777         {0x1c50, 0x1c7f, 1},
3778     },
3779 }
3780
3781 var _Old_Italic = &RangeTable{
3782     R16: []Range16{},
3783     R32: []Range32{
3784         {0x10300, 0x1031e, 1},
3785         {0x10320, 0x10323, 1},
3786     },
3787 }
3788
3789 var _Old_Persian = &RangeTable{
3790     R16: []Range16{},
3791     R32: []Range32{
3792         {0x103a0, 0x103c3, 1},
3793         {0x103c8, 0x103d5, 1},
3794     },

```

```

3795 }
3796
3797 var _Old_South_Arabian = &RangeTable{
3798     R16: []Range16{},
3799     R32: []Range32{
3800         {0x10a60, 0x10a7f, 1},
3801     },
3802 }
3803
3804 var _Old_Turkic = &RangeTable{
3805     R16: []Range16{},
3806     R32: []Range32{
3807         {0x10c00, 0x10c48, 1},
3808     },
3809 }
3810
3811 var _Oriya = &RangeTable{
3812     R16: []Range16{
3813         {0x0b01, 0x0b03, 1},
3814         {0x0b05, 0x0b0c, 1},
3815         {0x0b0f, 0x0b10, 1},
3816         {0x0b13, 0x0b28, 1},
3817         {0x0b2a, 0x0b30, 1},
3818         {0x0b32, 0x0b33, 1},
3819         {0x0b35, 0x0b39, 1},
3820         {0x0b3c, 0x0b44, 1},
3821         {0x0b47, 0x0b48, 1},
3822         {0x0b4b, 0x0b4d, 1},
3823         {0x0b56, 0x0b57, 1},
3824         {0x0b5c, 0x0b5d, 1},
3825         {0x0b5f, 0x0b63, 1},
3826         {0x0b66, 0x0b77, 1},
3827     },
3828 }
3829
3830 var _Osmanya = &RangeTable{
3831     R16: []Range16{},
3832     R32: []Range32{
3833         {0x10480, 0x1049d, 1},
3834         {0x104a0, 0x104a9, 1},
3835     },
3836 }
3837
3838 var _Phags_Pa = &RangeTable{
3839     R16: []Range16{
3840         {0xa840, 0xa877, 1},
3841     },
3842 }
3843

```

```

3844 var _Phoenician = &RangeTable{
3845     R16: []Range16{},
3846     R32: []Range32{
3847         {0x10900, 0x1091b, 1},
3848         {0x1091f, 0x1091f, 1},
3849     },
3850 }
3851
3852 var _Rejang = &RangeTable{
3853     R16: []Range16{
3854         {0xa930, 0xa953, 1},
3855         {0xa95f, 0xa95f, 1},
3856     },
3857 }
3858
3859 var _Runic = &RangeTable{
3860     R16: []Range16{
3861         {0x16a0, 0x16ea, 1},
3862         {0x16ee, 0x16f0, 1},
3863     },
3864 }
3865
3866 var _Samaritan = &RangeTable{
3867     R16: []Range16{
3868         {0x0800, 0x082d, 1},
3869         {0x0830, 0x083e, 1},
3870     },
3871 }
3872
3873 var _Saurashtra = &RangeTable{
3874     R16: []Range16{
3875         {0xa880, 0xa8c4, 1},
3876         {0xa8ce, 0xa8d9, 1},
3877     },
3878 }
3879
3880 var _Shavian = &RangeTable{
3881     R16: []Range16{},
3882     R32: []Range32{
3883         {0x10450, 0x1047f, 1},
3884     },
3885 }
3886
3887 var _Sinhala = &RangeTable{
3888     R16: []Range16{
3889         {0x0d82, 0x0d83, 1},
3890         {0x0d85, 0x0d96, 1},
3891         {0x0d9a, 0x0db1, 1},
3892         {0x0db3, 0x0dbb, 1},

```

```

3893             {0x0dbd, 0x0dbd, 1},
3894             {0x0dc0, 0x0dc6, 1},
3895             {0x0dca, 0x0dca, 1},
3896             {0x0dcf, 0x0dd4, 1},
3897             {0x0dd6, 0x0dd6, 1},
3898             {0x0dd8, 0x0ddf, 1},
3899             {0x0df2, 0x0df4, 1},
3900         },
3901     }
3902
3903     var _Sundanese = &RangeTable{
3904         R16: []Range16{
3905             {0x1b80, 0x1baa, 1},
3906             {0x1bae, 0x1bb9, 1},
3907         },
3908     }
3909
3910     var _Syloti_Nagri = &RangeTable{
3911         R16: []Range16{
3912             {0xa800, 0xa82b, 1},
3913         },
3914     }
3915
3916     var _Syriac = &RangeTable{
3917         R16: []Range16{
3918             {0x0700, 0x070d, 1},
3919             {0x070f, 0x074a, 1},
3920             {0x074d, 0x074f, 1},
3921         },
3922     }
3923
3924     var _Tagalog = &RangeTable{
3925         R16: []Range16{
3926             {0x1700, 0x170c, 1},
3927             {0x170e, 0x1714, 1},
3928         },
3929     }
3930
3931     var _Tagbanwa = &RangeTable{
3932         R16: []Range16{
3933             {0x1760, 0x176c, 1},
3934             {0x176e, 0x1770, 1},
3935             {0x1772, 0x1773, 1},
3936         },
3937     }
3938
3939     var _Tai_Le = &RangeTable{
3940         R16: []Range16{
3941             {0x1950, 0x196d, 1},
3942             {0x1970, 0x1974, 1},

```

```

3943     },
3944 }
3945
3946 var _Tai_Tham = &RangeTable{
3947     R16: []Range16{
3948         {0x1a20, 0x1a5e, 1},
3949         {0x1a60, 0x1a7c, 1},
3950         {0x1a7f, 0x1a89, 1},
3951         {0x1a90, 0x1a99, 1},
3952         {0x1aa0, 0x1aad, 1},
3953     },
3954 }
3955
3956 var _Tai_Viet = &RangeTable{
3957     R16: []Range16{
3958         {0xaa80, 0xaac2, 1},
3959         {0xaadb, 0xaadf, 1},
3960     },
3961 }
3962
3963 var _Tamil = &RangeTable{
3964     R16: []Range16{
3965         {0x0b82, 0x0b83, 1},
3966         {0x0b85, 0x0b8a, 1},
3967         {0x0b8e, 0x0b90, 1},
3968         {0x0b92, 0x0b95, 1},
3969         {0x0b99, 0x0b9a, 1},
3970         {0x0b9c, 0x0b9c, 1},
3971         {0x0b9e, 0x0b9f, 1},
3972         {0x0ba3, 0x0ba4, 1},
3973         {0x0ba8, 0x0baa, 1},
3974         {0x0bae, 0x0bb9, 1},
3975         {0x0bbe, 0x0bc2, 1},
3976         {0x0bc6, 0x0bc8, 1},
3977         {0x0bca, 0x0bcd, 1},
3978         {0x0bd0, 0x0bd0, 1},
3979         {0x0bd7, 0x0bd7, 1},
3980         {0x0be6, 0x0bfa, 1},
3981     },
3982 }
3983
3984 var _Telugu = &RangeTable{
3985     R16: []Range16{
3986         {0x0c01, 0x0c03, 1},
3987         {0x0c05, 0x0c0c, 1},
3988         {0x0c0e, 0x0c10, 1},
3989         {0x0c12, 0x0c28, 1},
3990         {0x0c2a, 0x0c33, 1},
3991         {0x0c35, 0x0c39, 1},

```

```

3992             {0x0c3d, 0x0c44, 1},
3993             {0x0c46, 0x0c48, 1},
3994             {0x0c4a, 0x0c4d, 1},
3995             {0x0c55, 0x0c56, 1},
3996             {0x0c58, 0x0c59, 1},
3997             {0x0c60, 0x0c63, 1},
3998             {0x0c66, 0x0c6f, 1},
3999             {0x0c78, 0x0c7f, 1},
4000         },
4001     }
4002
4003     var _Thaana = &RangeTable{
4004         R16: []Range16{
4005             {0x0780, 0x07b1, 1},
4006         },
4007     }
4008
4009     var _Thai = &RangeTable{
4010         R16: []Range16{
4011             {0x0e01, 0x0e3a, 1},
4012             {0x0e40, 0x0e5b, 1},
4013         },
4014     }
4015
4016     var _Tibetan = &RangeTable{
4017         R16: []Range16{
4018             {0x0f00, 0x0f47, 1},
4019             {0x0f49, 0x0f6c, 1},
4020             {0x0f71, 0x0f97, 1},
4021             {0x0f99, 0x0fbc, 1},
4022             {0x0fbe, 0x0fcc, 1},
4023             {0x0fce, 0x0fd4, 1},
4024             {0x0fd9, 0x0fda, 1},
4025         },
4026     }
4027
4028     var _Tifinagh = &RangeTable{
4029         R16: []Range16{
4030             {0x2d30, 0x2d65, 1},
4031             {0x2d6f, 0x2d70, 1},
4032             {0x2d7f, 0x2d7f, 1},
4033         },
4034     }
4035
4036     var _Ugaritic = &RangeTable{
4037         R16: []Range16{},
4038         R32: []Range32{
4039             {0x10380, 0x1039d, 1},
4040             {0x1039f, 0x1039f, 1},

```

```

4041     },
4042 }
4043
4044 var _Vai = &RangeTable{
4045     R16: []Range16{
4046         {0xa500, 0xa62b, 1},
4047     },
4048 }
4049
4050 var _Yi = &RangeTable{
4051     R16: []Range16{
4052         {0xa000, 0xa48c, 1},
4053         {0xa490, 0xa4c6, 1},
4054     },
4055 }
4056
4057 // The following variables are of type *RangeTable:
4058 var (
4059     Arabic           = _Arabic           //
4060     Armenian        = _Armenian         //
4061     Avestan         = _Avestan          //
4062     Balinese        = _Balinese         //
4063     Bamum           = _Bamum            //
4064     Batak           = _Batak            //
4065     Bengali         = _Bengali          //
4066     Bopomofo       = _Bopomofo         //
4067     Brahmi          = _Brahmi           //
4068     Braille         = _Braille          //
4069     Buginese        = _Buginese         //
4070     Buhid           = _Buhid            //
4071     Canadian_Aboriginal = _Canadian_Aboriginal //
4072     Carian          = _Carian           //
4073     Cham            = _Cham             //
4074     Cherokee        = _Cherokee         //
4075     Common          = _Common           //
4076     Coptic          = _Coptic           //
4077     Cuneiform       = _Cuneiform        //
4078     Cypriot         = _Cypriot          //
4079     Cyrillic        = _Cyrillic         //
4080     Deseret         = _Deseret          //
4081     Devanagari      = _Devanagari       //
4082     Egyptian_Hieroglyphs = _Egyptian_Hieroglyphs //
4083     Ethiopic        = _Ethiopic         //
4084     Georgian        = _Georgian         //
4085     Glagolitic      = _Glagolitic       //
4086     Gothic          = _Gothic           //
4087     Greek           = _Greek            //
4088     Gujarati        = _Gujarati         //
4089     Gurmukhi        = _Gurmukhi        //
4090     Han             = _Han               //

```

4091	Hangul	=	_Hangul	//
4092	Hanunoo	=	_Hanunoo	//
4093	Hebrew	=	_Hebrew	//
4094	Hiragana	=	_Hiragana	//
4095	Imperial_Aramaic	=	_Imperial_Aramaic	//
4096	Inherited	=	_Inherited	//
4097	Inscriptional_Pahlavi	=	_Inscriptional_Pahlavi	//
4098	Inscriptional_Parthian	=	_Inscriptional_Parthian	//
4099	Javanese	=	_Javanese	//
4100	Kaithi	=	_Kaithi	//
4101	Kannada	=	_Kannada	//
4102	Katakana	=	_Katakana	//
4103	Kayah_Li	=	_Kayah_Li	//
4104	Kharoshthi	=	_Kharoshthi	//
4105	Khmer	=	_Khmer	//
4106	Lao	=	_Lao	//
4107	Latin	=	_Latin	//
4108	Lepcha	=	_Lepcha	//
4109	Limbu	=	_Limbu	//
4110	Linear_B	=	_Linear_B	//
4111	Lisu	=	_Lisu	//
4112	Lycian	=	_Lycian	//
4113	Lydian	=	_Lydian	//
4114	Malayalam	=	_Malayalam	//
4115	Mandaic	=	_Mandaic	//
4116	Meetei_Mayek	=	_Meetei_Mayek	//
4117	Mongolian	=	_Mongolian	//
4118	Myanmar	=	_Myanmar	//
4119	New_Tai_Lue	=	_New_Tai_Lue	//
4120	Nko	=	_Nko	//
4121	Ogham	=	_Ogham	//
4122	Ol_Chiki	=	_Ol_Chiki	//
4123	Old_Italic	=	_Old_Italic	//
4124	Old_Persian	=	_Old_Persian	//
4125	Old_South_Arabian	=	_Old_South_Arabian	//
4126	Old_Turkic	=	_Old_Turkic	//
4127	Oriya	=	_Oriya	//
4128	Osmanya	=	_Osmanya	//
4129	Phags_Pa	=	_Phags_Pa	//
4130	Phoenician	=	_Phoenician	//
4131	Rejang	=	_Rejang	//
4132	Runic	=	_Runic	//
4133	Samaritan	=	_Samaritan	//
4134	Saurashtra	=	_Saurashtra	//
4135	Shavian	=	_Shavian	//
4136	Sinhala	=	_Sinhala	//
4137	Sundanese	=	_Sundanese	//
4138	Syloti_Nagri	=	_Syloti_Nagri	//
4139	Syriac	=	_Syriac	//

```

4140         Tagalog           = _Tagalog           //
4141         Tagbanwa          = _Tagbanwa          //
4142         Tai_Le             = _Tai_Le             //
4143         Tai_Tham           = _Tai_Tham           //
4144         Tai_Viet           = _Tai_Viet           //
4145         Tamil              = _Tamil              //
4146         Telugu             = _Telugu             //
4147         Thaana             = _Thaana             //
4148         Thai               = _Thai               //
4149         Tibetan            = _Tibetan            //
4150         Tifinagh           = _Tifinagh           //
4151         Ugaritic           = _Ugaritic           //
4152         Vai                = _Vai                //
4153         Yi                 = _Yi                 //
4154     )
4155
4156 // Generated by running
4157 //     maketables --props=all --url=http://www.unicode.org/
4158 // DO NOT EDIT
4159
4160 // Properties is the set of Unicode property tables.
4161 var Properties = map[string]*RangeTable{
4162     "ASCII_Hex_Digit":      ASCII_Hex_Digi
4163     "Bidi_Control":         Bidi_Control,
4164     "Dash":                 Dash,
4165     "Deprecated":          Deprecated,
4166     "Diacritic":            Diacritic,
4167     "Extender":             Extender,
4168     "Hex_Digit":            Hex_Digit,
4169     "Hyphen":               Hyphen,
4170     "IDS_Binary_Operator":  IDS_Binary_Ope
4171     "IDS_Tertiary_Operator": IDS_Tertiary_Op
4172     "Ideographic":         Ideographic,
4173     "Join_Control":        Join_Control,
4174     "Logical_Order_Exception": Logical_Order_
4175     "Noncharacter_Code_Point": Noncharacter_C
4176     "Other_Alphabetic":     Other_Alphabet
4177     "Other_Default_Ignorable_Code_Point": Other_Default_
4178     "Other_Grapheme_Extend": Other_Grapheme
4179     "Other_ID_Continue":    Other_ID_Conti
4180     "Other_ID_Start":       Other_ID_Start
4181     "Other_Lowercase":      Other_Lowercas
4182     "Other_Math":           Other_Math,
4183     "Other_Uppercase":      Other_Uppercas
4184     "Pattern_Syntax":       Pattern_Syntax
4185     "Pattern_White_Space":  Pattern_White_
4186     "Quotation_Mark":       Quotation_Mark
4187     "Radical":              Radical,
4188     "STerm":                 STerm,

```

```

4189         "Soft_Dotted":                Soft_Dotted,
4190         "Terminal_Punctuation":        Terminal_Punct
4191         "Unified_Ideograph":           Unified_Ideogr
4192         "Variation_Selector":          Variation_Sele
4193         "White_Space":                  White_Space,
4194     }
4195
4196     var _ASCII_Hex_Digit = &RangeTable{
4197         R16: []Range16{
4198             {0x0030, 0x0039, 1},
4199             {0x0041, 0x0046, 1},
4200             {0x0061, 0x0066, 1},
4201         },
4202     }
4203
4204     var _Bidi_Control = &RangeTable{
4205         R16: []Range16{
4206             {0x200e, 0x200f, 1},
4207             {0x202a, 0x202e, 1},
4208         },
4209     }
4210
4211     var _Dash = &RangeTable{
4212         R16: []Range16{
4213             {0x002d, 0x002d, 1},
4214             {0x058a, 0x058a, 1},
4215             {0x05be, 0x05be, 1},
4216             {0x1400, 0x1400, 1},
4217             {0x1806, 0x1806, 1},
4218             {0x2010, 0x2015, 1},
4219             {0x2053, 0x2053, 1},
4220             {0x207b, 0x207b, 1},
4221             {0x208b, 0x208b, 1},
4222             {0x2212, 0x2212, 1},
4223             {0x2e17, 0x2e17, 1},
4224             {0x2e1a, 0x2e1a, 1},
4225             {0x301c, 0x301c, 1},
4226             {0x3030, 0x3030, 1},
4227             {0x30a0, 0x30a0, 1},
4228             {0xfe31, 0xfe32, 1},
4229             {0xfe58, 0xfe58, 1},
4230             {0xfe63, 0xfe63, 1},
4231             {0xff0d, 0xff0d, 1},
4232         },
4233     }
4234
4235     var _Deprecated = &RangeTable{
4236         R16: []Range16{
4237             {0x0149, 0x0149, 1},
4238             {0x0673, 0x0673, 1},

```

```

4239             {0x0f77, 0x0f77, 1},
4240             {0x0f79, 0x0f79, 1},
4241             {0x17a3, 0x17a4, 1},
4242             {0x206a, 0x206f, 1},
4243             {0x2329, 0x232a, 1},
4244         },
4245         R32: []Range32{
4246             {0xe0001, 0xe0001, 1},
4247             {0xe0020, 0xe007f, 1},
4248         },
4249     }
4250
4251     var _Diacritic = &RangeTable{
4252         R16: []Range16{
4253             {0x005e, 0x005e, 1},
4254             {0x0060, 0x0060, 1},
4255             {0x00a8, 0x00a8, 1},
4256             {0x00af, 0x00af, 1},
4257             {0x00b4, 0x00b4, 1},
4258             {0x00b7, 0x00b8, 1},
4259             {0x02b0, 0x034e, 1},
4260             {0x0350, 0x0357, 1},
4261             {0x035d, 0x0362, 1},
4262             {0x0374, 0x0375, 1},
4263             {0x037a, 0x037a, 1},
4264             {0x0384, 0x0385, 1},
4265             {0x0483, 0x0487, 1},
4266             {0x0559, 0x0559, 1},
4267             {0x0591, 0x05a1, 1},
4268             {0x05a3, 0x05bd, 1},
4269             {0x05bf, 0x05bf, 1},
4270             {0x05c1, 0x05c2, 1},
4271             {0x05c4, 0x05c4, 1},
4272             {0x064b, 0x0652, 1},
4273             {0x0657, 0x0658, 1},
4274             {0x06df, 0x06e0, 1},
4275             {0x06e5, 0x06e6, 1},
4276             {0x06ea, 0x06ec, 1},
4277             {0x0730, 0x074a, 1},
4278             {0x07a6, 0x07b0, 1},
4279             {0x07eb, 0x07f5, 1},
4280             {0x0818, 0x0819, 1},
4281             {0x093c, 0x093c, 1},
4282             {0x094d, 0x094d, 1},
4283             {0x0951, 0x0954, 1},
4284             {0x0971, 0x0971, 1},
4285             {0x09bc, 0x09bc, 1},
4286             {0x09cd, 0x09cd, 1},
4287             {0x0a3c, 0x0a3c, 1},

```

4288 {0x0a4d, 0x0a4d, 1},
4289 {0x0abc, 0x0abc, 1},
4290 {0x0acd, 0x0acd, 1},
4291 {0x0b3c, 0x0b3c, 1},
4292 {0x0b4d, 0x0b4d, 1},
4293 {0x0bcd, 0x0bcd, 1},
4294 {0x0c4d, 0x0c4d, 1},
4295 {0x0cbc, 0x0cbc, 1},
4296 {0x0ccd, 0x0ccd, 1},
4297 {0x0d4d, 0x0d4d, 1},
4298 {0x0dca, 0x0dca, 1},
4299 {0x0e47, 0x0e4c, 1},
4300 {0x0e4e, 0x0e4e, 1},
4301 {0x0ec8, 0x0ecc, 1},
4302 {0x0f18, 0x0f19, 1},
4303 {0x0f35, 0x0f35, 1},
4304 {0x0f37, 0x0f37, 1},
4305 {0x0f39, 0x0f39, 1},
4306 {0x0f3e, 0x0f3f, 1},
4307 {0x0f82, 0x0f84, 1},
4308 {0x0f86, 0x0f87, 1},
4309 {0x0fc6, 0x0fc6, 1},
4310 {0x1037, 0x1037, 1},
4311 {0x1039, 0x103a, 1},
4312 {0x1087, 0x108d, 1},
4313 {0x108f, 0x108f, 1},
4314 {0x109a, 0x109b, 1},
4315 {0x17c9, 0x17d3, 1},
4316 {0x17dd, 0x17dd, 1},
4317 {0x1939, 0x193b, 1},
4318 {0x1a75, 0x1a7c, 1},
4319 {0x1a7f, 0x1a7f, 1},
4320 {0x1b34, 0x1b34, 1},
4321 {0x1b44, 0x1b44, 1},
4322 {0x1b6b, 0x1b73, 1},
4323 {0x1baa, 0x1baa, 1},
4324 {0x1c36, 0x1c37, 1},
4325 {0x1c78, 0x1c7d, 1},
4326 {0x1cd0, 0x1ce8, 1},
4327 {0x1ced, 0x1ced, 1},
4328 {0x1d2c, 0x1d6a, 1},
4329 {0x1dc4, 0x1dcf, 1},
4330 {0x1dfd, 0x1dff, 1},
4331 {0x1fbd, 0x1fbd, 1},
4332 {0x1fbf, 0x1fc1, 1},
4333 {0x1fcd, 0x1fcf, 1},
4334 {0x1fdd, 0x1fdf, 1},
4335 {0x1fed, 0x1fef, 1},
4336 {0x1ffd, 0x1ffe, 1},

```

4337         {0x2cef, 0x2cf1, 1},
4338         {0x2e2f, 0x2e2f, 1},
4339         {0x302a, 0x302f, 1},
4340         {0x3099, 0x309c, 1},
4341         {0x30fc, 0x30fc, 1},
4342         {0xa66f, 0xa66f, 1},
4343         {0xa67c, 0xa67d, 1},
4344         {0xa67f, 0xa67f, 1},
4345         {0xa6f0, 0xa6f1, 1},
4346         {0xa717, 0xa721, 1},
4347         {0xa788, 0xa788, 1},
4348         {0xa8c4, 0xa8c4, 1},
4349         {0xa8e0, 0xa8f1, 1},
4350         {0xa92b, 0xa92e, 1},
4351         {0xa953, 0xa953, 1},
4352         {0xa9b3, 0xa9b3, 1},
4353         {0xa9c0, 0xa9c0, 1},
4354         {0xaa7b, 0xaa7b, 1},
4355         {0xaabf, 0xaac2, 1},
4356         {0xabec, 0xabed, 1},
4357         {0xfb1e, 0xfb1e, 1},
4358         {0xfe20, 0xfe26, 1},
4359         {0xff3e, 0xff3e, 1},
4360         {0xff40, 0xff40, 1},
4361         {0xff70, 0xff70, 1},
4362         {0xff9e, 0xff9f, 1},
4363         {0xffe3, 0xffe3, 1},
4364     },
4365     R32: []Range32{
4366         {0x110b9, 0x110ba, 1},
4367         {0x1d167, 0x1d169, 1},
4368         {0x1d16d, 0x1d172, 1},
4369         {0x1d17b, 0x1d182, 1},
4370         {0x1d185, 0x1d18b, 1},
4371         {0x1d1aa, 0x1d1ad, 1},
4372     },
4373 }
4374
4375 var _Extender = &RangeTable{
4376     R16: []Range16{
4377         {0x00b7, 0x00b7, 1},
4378         {0x02d0, 0x02d1, 1},
4379         {0x0640, 0x0640, 1},
4380         {0x07fa, 0x07fa, 1},
4381         {0x0e46, 0x0e46, 1},
4382         {0x0ec6, 0x0ec6, 1},
4383         {0x1843, 0x1843, 1},
4384         {0x1aa7, 0x1aa7, 1},
4385         {0x1c36, 0x1c36, 1},
4386         {0x1c7b, 0x1c7b, 1},

```

```

4387         {0x3005, 0x3005, 1},
4388         {0x3031, 0x3035, 1},
4389         {0x309d, 0x309e, 1},
4390         {0x30fc, 0x30fe, 1},
4391         {0xa015, 0xa015, 1},
4392         {0xa60c, 0xa60c, 1},
4393         {0xa9cf, 0xa9cf, 1},
4394         {0xaa70, 0xaa70, 1},
4395         {0xaadd, 0xaadd, 1},
4396         {0xff70, 0xff70, 1},
4397     },
4398 }
4399
4400 var _Hex_Digit = &RangeTable{
4401     R16: []Range16{
4402         {0x0030, 0x0039, 1},
4403         {0x0041, 0x0046, 1},
4404         {0x0061, 0x0066, 1},
4405         {0xff10, 0xff19, 1},
4406         {0xff21, 0xff26, 1},
4407         {0xff41, 0xff46, 1},
4408     },
4409 }
4410
4411 var _Hyphen = &RangeTable{
4412     R16: []Range16{
4413         {0x002d, 0x002d, 1},
4414         {0x00ad, 0x00ad, 1},
4415         {0x058a, 0x058a, 1},
4416         {0x1806, 0x1806, 1},
4417         {0x2010, 0x2011, 1},
4418         {0x2e17, 0x2e17, 1},
4419         {0x30fb, 0x30fb, 1},
4420         {0xfe63, 0xfe63, 1},
4421         {0xff0d, 0xff0d, 1},
4422         {0xff65, 0xff65, 1},
4423     },
4424 }
4425
4426 var _IDS_Binary_Operator = &RangeTable{
4427     R16: []Range16{
4428         {0x2ff0, 0x2ff1, 1},
4429         {0x2ff4, 0x2ffb, 1},
4430     },
4431 }
4432
4433 var _IDS_Ternary_Operator = &RangeTable{
4434     R16: []Range16{
4435         {0x2ff2, 0x2ff3, 1},

```

```

4436     },
4437 }
4438
4439 var _Ideographic = &RangeTable{
4440     R16: []Range16{
4441         {0x3006, 0x3007, 1},
4442         {0x3021, 0x3029, 1},
4443         {0x3038, 0x303a, 1},
4444         {0x3400, 0x4db5, 1},
4445         {0x4e00, 0x9fcb, 1},
4446         {0xf900, 0xfa2d, 1},
4447         {0xfa30, 0xfa6d, 1},
4448         {0xfa70, 0xfad9, 1},
4449     },
4450     R32: []Range32{
4451         {0x20000, 0x2a6d6, 1},
4452         {0x2a700, 0x2b734, 1},
4453         {0x2b740, 0x2b81d, 1},
4454         {0x2f800, 0x2fa1d, 1},
4455     },
4456 }
4457
4458 var _Join_Control = &RangeTable{
4459     R16: []Range16{
4460         {0x200c, 0x200d, 1},
4461     },
4462 }
4463
4464 var _Logical_Order_Exception = &RangeTable{
4465     R16: []Range16{
4466         {0x0e40, 0x0e44, 1},
4467         {0x0ec0, 0x0ec4, 1},
4468         {0xaab5, 0xaab6, 1},
4469         {0xaab9, 0xaab9, 1},
4470         {0xaabb, 0xaabc, 1},
4471     },
4472 }
4473
4474 var _Noncharacter_Code_Point = &RangeTable{
4475     R16: []Range16{
4476         {0xfdd0, 0xfdef, 1},
4477         {0xfffe, 0xffff, 1},
4478     },
4479     R32: []Range32{
4480         {0x1fffe, 0x1ffff, 1},
4481         {0x2fffe, 0x2ffff, 1},
4482         {0x3fffe, 0x3ffff, 1},
4483         {0x4fffe, 0x4ffff, 1},
4484         {0x5fffe, 0x5ffff, 1},

```

```

4485         {0x6fffe, 0x6ffff, 1},
4486         {0x7fffe, 0x7ffff, 1},
4487         {0x8fffe, 0x8ffff, 1},
4488         {0x9fffe, 0x9ffff, 1},
4489         {0xafffe, 0xaffff, 1},
4490         {0xbfffe, 0xbffff, 1},
4491         {0xcfffe, 0xcffff, 1},
4492         {0xdfffe, 0xdffff, 1},
4493         {0xefffe, 0xeffff, 1},
4494         {0xffffe, 0xfffff, 1},
4495         {0x10fffe, 0x10ffff, 1},
4496     },
4497 }
4498
4499 var _Other_Alphabetic = &RangeTable{
4500     R16: []Range16{
4501         {0x0345, 0x0345, 1},
4502         {0x05b0, 0x05bd, 1},
4503         {0x05bf, 0x05bf, 1},
4504         {0x05c1, 0x05c2, 1},
4505         {0x05c4, 0x05c5, 1},
4506         {0x05c7, 0x05c7, 1},
4507         {0x0610, 0x061a, 1},
4508         {0x064b, 0x0657, 1},
4509         {0x0659, 0x065f, 1},
4510         {0x0670, 0x0670, 1},
4511         {0x06d6, 0x06dc, 1},
4512         {0x06e1, 0x06e4, 1},
4513         {0x06e7, 0x06e8, 1},
4514         {0x06ed, 0x06ed, 1},
4515         {0x0711, 0x0711, 1},
4516         {0x0730, 0x073f, 1},
4517         {0x07a6, 0x07b0, 1},
4518         {0x0816, 0x0817, 1},
4519         {0x081b, 0x0823, 1},
4520         {0x0825, 0x0827, 1},
4521         {0x0829, 0x082c, 1},
4522         {0x0900, 0x0903, 1},
4523         {0x093a, 0x093b, 1},
4524         {0x093e, 0x094c, 1},
4525         {0x094e, 0x094f, 1},
4526         {0x0955, 0x0957, 1},
4527         {0x0962, 0x0963, 1},
4528         {0x0981, 0x0983, 1},
4529         {0x09be, 0x09c4, 1},
4530         {0x09c7, 0x09c8, 1},
4531         {0x09cb, 0x09cc, 1},
4532         {0x09d7, 0x09d7, 1},
4533         {0x09e2, 0x09e3, 1},
4534         {0x0a01, 0x0a03, 1},

```

4535	{0x0a3e, 0x0a42, 1},
4536	{0x0a47, 0x0a48, 1},
4537	{0x0a4b, 0x0a4c, 1},
4538	{0x0a51, 0x0a51, 1},
4539	{0x0a70, 0x0a71, 1},
4540	{0x0a75, 0x0a75, 1},
4541	{0x0a81, 0x0a83, 1},
4542	{0x0abe, 0x0ac5, 1},
4543	{0x0ac7, 0x0ac9, 1},
4544	{0x0acb, 0x0acc, 1},
4545	{0x0ae2, 0x0ae3, 1},
4546	{0x0b01, 0x0b03, 1},
4547	{0x0b3e, 0x0b44, 1},
4548	{0x0b47, 0x0b48, 1},
4549	{0x0b4b, 0x0b4c, 1},
4550	{0x0b56, 0x0b57, 1},
4551	{0x0b62, 0x0b63, 1},
4552	{0x0b82, 0x0b82, 1},
4553	{0x0bbe, 0x0bc2, 1},
4554	{0x0bc6, 0x0bc8, 1},
4555	{0x0bca, 0x0bcc, 1},
4556	{0x0bd7, 0x0bd7, 1},
4557	{0x0c01, 0x0c03, 1},
4558	{0x0c3e, 0x0c44, 1},
4559	{0x0c46, 0x0c48, 1},
4560	{0x0c4a, 0x0c4c, 1},
4561	{0x0c55, 0x0c56, 1},
4562	{0x0c62, 0x0c63, 1},
4563	{0x0c82, 0x0c83, 1},
4564	{0x0cbe, 0x0cc4, 1},
4565	{0x0cc6, 0x0cc8, 1},
4566	{0x0cca, 0x0ccc, 1},
4567	{0x0cd5, 0x0cd6, 1},
4568	{0x0ce2, 0x0ce3, 1},
4569	{0x0d02, 0x0d03, 1},
4570	{0x0d3e, 0x0d44, 1},
4571	{0x0d46, 0x0d48, 1},
4572	{0x0d4a, 0x0d4c, 1},
4573	{0x0d57, 0x0d57, 1},
4574	{0x0d62, 0x0d63, 1},
4575	{0x0d82, 0x0d83, 1},
4576	{0x0dcf, 0x0dd4, 1},
4577	{0x0dd6, 0x0dd6, 1},
4578	{0x0dd8, 0x0ddf, 1},
4579	{0x0df2, 0x0df3, 1},
4580	{0x0e31, 0x0e31, 1},
4581	{0x0e34, 0x0e3a, 1},
4582	{0x0e4d, 0x0e4d, 1},
4583	{0x0eb1, 0x0eb1, 1},

4584	{0x0eb4, 0x0eb9, 1},
4585	{0x0ebb, 0x0ebc, 1},
4586	{0x0ecd, 0x0ecd, 1},
4587	{0x0f71, 0x0f81, 1},
4588	{0x0f8d, 0x0f97, 1},
4589	{0x0f99, 0x0fbc, 1},
4590	{0x102b, 0x1036, 1},
4591	{0x1038, 0x1038, 1},
4592	{0x103b, 0x103e, 1},
4593	{0x1056, 0x1059, 1},
4594	{0x105e, 0x1060, 1},
4595	{0x1062, 0x1062, 1},
4596	{0x1067, 0x1068, 1},
4597	{0x1071, 0x1074, 1},
4598	{0x1082, 0x1086, 1},
4599	{0x109c, 0x109d, 1},
4600	{0x135f, 0x135f, 1},
4601	{0x1712, 0x1713, 1},
4602	{0x1732, 0x1733, 1},
4603	{0x1752, 0x1753, 1},
4604	{0x1772, 0x1773, 1},
4605	{0x17b6, 0x17c8, 1},
4606	{0x18a9, 0x18a9, 1},
4607	{0x1920, 0x192b, 1},
4608	{0x1930, 0x1938, 1},
4609	{0x19b0, 0x19c0, 1},
4610	{0x19c8, 0x19c9, 1},
4611	{0x1a17, 0x1a1b, 1},
4612	{0x1a55, 0x1a5e, 1},
4613	{0x1a61, 0x1a74, 1},
4614	{0x1b00, 0x1b04, 1},
4615	{0x1b35, 0x1b43, 1},
4616	{0x1b80, 0x1b82, 1},
4617	{0x1ba1, 0x1ba9, 1},
4618	{0x1be7, 0x1bf1, 1},
4619	{0x1c24, 0x1c35, 1},
4620	{0x1cf2, 0x1cf2, 1},
4621	{0x24b6, 0x24e9, 1},
4622	{0x2de0, 0x2dff, 1},
4623	{0xa823, 0xa827, 1},
4624	{0xa880, 0xa881, 1},
4625	{0xa8b4, 0xa8c3, 1},
4626	{0xa926, 0xa92a, 1},
4627	{0xa947, 0xa952, 1},
4628	{0xa980, 0xa983, 1},
4629	{0xa9b4, 0xa9bf, 1},
4630	{0xaa29, 0xaa36, 1},
4631	{0xaa43, 0xaa43, 1},
4632	{0xaa4c, 0xaa4d, 1},

```

4633             {0xaab0, 0xaab0, 1},
4634             {0xaab2, 0xaab4, 1},
4635             {0xaab7, 0xaab8, 1},
4636             {0xaabe, 0xaabe, 1},
4637             {0xabe3, 0xabea, 1},
4638             {0xfb1e, 0xfb1e, 1},
4639         },
4640         R32: []Range32{
4641             {0x10a01, 0x10a03, 1},
4642             {0x10a05, 0x10a06, 1},
4643             {0x10a0c, 0x10a0f, 1},
4644             {0x11000, 0x11002, 1},
4645             {0x11038, 0x11045, 1},
4646             {0x11082, 0x11082, 1},
4647             {0x110b0, 0x110b8, 1},
4648         },
4649     }
4650
4651     var _Other_Default_Ignorable_Code_Point = &RangeTable{
4652         R16: []Range16{
4653             {0x034f, 0x034f, 1},
4654             {0x115f, 0x1160, 1},
4655             {0x2065, 0x2069, 1},
4656             {0x3164, 0x3164, 1},
4657             {0xffa0, 0xffa0, 1},
4658             {0xffff0, 0xffff8, 1},
4659         },
4660         R32: []Range32{
4661             {0xe0000, 0xe0000, 1},
4662             {0xe0002, 0xe001f, 1},
4663             {0xe0080, 0xe00ff, 1},
4664             {0xe01f0, 0xe0fff, 1},
4665         },
4666     }
4667
4668     var _Other_Grapheme_Extend = &RangeTable{
4669         R16: []Range16{
4670             {0x09be, 0x09be, 1},
4671             {0x09d7, 0x09d7, 1},
4672             {0x0b3e, 0x0b3e, 1},
4673             {0x0b57, 0x0b57, 1},
4674             {0x0bbe, 0x0bbe, 1},
4675             {0x0bd7, 0x0bd7, 1},
4676             {0x0cc2, 0x0cc2, 1},
4677             {0x0cd5, 0x0cd6, 1},
4678             {0x0d3e, 0x0d3e, 1},
4679             {0x0d57, 0x0d57, 1},
4680             {0x0dcf, 0x0dcf, 1},
4681             {0x0ddf, 0x0ddf, 1},
4682             {0x200c, 0x200d, 1},

```

```

4683             {0xff9e, 0xff9f, 1},
4684         },
4685         R32: []Range32{
4686             {0x1d165, 0x1d165, 1},
4687             {0x1d16e, 0x1d172, 1},
4688         },
4689     }
4690
4691     var _Other_ID_Continue = &RangeTable{
4692         R16: []Range16{
4693             {0x00b7, 0x00b7, 1},
4694             {0x0387, 0x0387, 1},
4695             {0x1369, 0x1371, 1},
4696             {0x19da, 0x19da, 1},
4697         },
4698     }
4699
4700     var _Other_ID_Start = &RangeTable{
4701         R16: []Range16{
4702             {0x2118, 0x2118, 1},
4703             {0x212e, 0x212e, 1},
4704             {0x309b, 0x309c, 1},
4705         },
4706     }
4707
4708     var _Other_Lowercase = &RangeTable{
4709         R16: []Range16{
4710             {0x02b0, 0x02b8, 1},
4711             {0x02c0, 0x02c1, 1},
4712             {0x02e0, 0x02e4, 1},
4713             {0x0345, 0x0345, 1},
4714             {0x037a, 0x037a, 1},
4715             {0x1d2c, 0x1d61, 1},
4716             {0x1d78, 0x1d78, 1},
4717             {0x1d9b, 0x1dbf, 1},
4718             {0x2090, 0x2094, 1},
4719             {0x2170, 0x217f, 1},
4720             {0x24d0, 0x24e9, 1},
4721             {0x2c7d, 0x2c7d, 1},
4722             {0xa770, 0xa770, 1},
4723         },
4724     }
4725
4726     var _Other_Math = &RangeTable{
4727         R16: []Range16{
4728             {0x005e, 0x005e, 1},
4729             {0x03d0, 0x03d2, 1},
4730             {0x03d5, 0x03d5, 1},
4731             {0x03f0, 0x03f1, 1},

```

4732	{0x03f4, 0x03f5, 1},
4733	{0x2016, 0x2016, 1},
4734	{0x2032, 0x2034, 1},
4735	{0x2040, 0x2040, 1},
4736	{0x2061, 0x2064, 1},
4737	{0x207d, 0x207e, 1},
4738	{0x208d, 0x208e, 1},
4739	{0x20d0, 0x20dc, 1},
4740	{0x20e1, 0x20e1, 1},
4741	{0x20e5, 0x20e6, 1},
4742	{0x20eb, 0x20ef, 1},
4743	{0x2102, 0x2102, 1},
4744	{0x2107, 0x2107, 1},
4745	{0x210a, 0x2113, 1},
4746	{0x2115, 0x2115, 1},
4747	{0x2119, 0x211d, 1},
4748	{0x2124, 0x2124, 1},
4749	{0x2128, 0x2129, 1},
4750	{0x212c, 0x212d, 1},
4751	{0x212f, 0x2131, 1},
4752	{0x2133, 0x2138, 1},
4753	{0x213c, 0x213f, 1},
4754	{0x2145, 0x2149, 1},
4755	{0x2195, 0x2199, 1},
4756	{0x219c, 0x219f, 1},
4757	{0x21a1, 0x21a2, 1},
4758	{0x21a4, 0x21a5, 1},
4759	{0x21a7, 0x21a7, 1},
4760	{0x21a9, 0x21ad, 1},
4761	{0x21b0, 0x21b1, 1},
4762	{0x21b6, 0x21b7, 1},
4763	{0x21bc, 0x21cd, 1},
4764	{0x21d0, 0x21d1, 1},
4765	{0x21d3, 0x21d3, 1},
4766	{0x21d5, 0x21db, 1},
4767	{0x21dd, 0x21dd, 1},
4768	{0x21e4, 0x21e5, 1},
4769	{0x23b4, 0x23b5, 1},
4770	{0x23b7, 0x23b7, 1},
4771	{0x23d0, 0x23d0, 1},
4772	{0x23e2, 0x23e2, 1},
4773	{0x25a0, 0x25a1, 1},
4774	{0x25ae, 0x25b6, 1},
4775	{0x25bc, 0x25c0, 1},
4776	{0x25c6, 0x25c7, 1},
4777	{0x25ca, 0x25cb, 1},
4778	{0x25cf, 0x25d3, 1},
4779	{0x25e2, 0x25e2, 1},
4780	{0x25e4, 0x25e4, 1},

```

4781         {0x25e7, 0x25ec, 1},
4782         {0x2605, 0x2606, 1},
4783         {0x2640, 0x2640, 1},
4784         {0x2642, 0x2642, 1},
4785         {0x2660, 0x2663, 1},
4786         {0x266d, 0x266e, 1},
4787         {0x27c5, 0x27c6, 1},
4788         {0x27e6, 0x27ef, 1},
4789         {0x2983, 0x2998, 1},
4790         {0x29d8, 0x29db, 1},
4791         {0x29fc, 0x29fd, 1},
4792         {0xfe61, 0xfe61, 1},
4793         {0xfe63, 0xfe63, 1},
4794         {0xfe68, 0xfe68, 1},
4795         {0xff3c, 0xff3c, 1},
4796         {0xff3e, 0xff3e, 1},
4797     },
4798     R32: []Range32{
4799         {0x1d400, 0x1d454, 1},
4800         {0x1d456, 0x1d49c, 1},
4801         {0x1d49e, 0x1d49f, 1},
4802         {0x1d4a2, 0x1d4a2, 1},
4803         {0x1d4a5, 0x1d4a6, 1},
4804         {0x1d4a9, 0x1d4ac, 1},
4805         {0x1d4ae, 0x1d4b9, 1},
4806         {0x1d4bb, 0x1d4bb, 1},
4807         {0x1d4bd, 0x1d4c3, 1},
4808         {0x1d4c5, 0x1d505, 1},
4809         {0x1d507, 0x1d50a, 1},
4810         {0x1d50d, 0x1d514, 1},
4811         {0x1d516, 0x1d51c, 1},
4812         {0x1d51e, 0x1d539, 1},
4813         {0x1d53b, 0x1d53e, 1},
4814         {0x1d540, 0x1d544, 1},
4815         {0x1d546, 0x1d546, 1},
4816         {0x1d54a, 0x1d550, 1},
4817         {0x1d552, 0x1d6a5, 1},
4818         {0x1d6a8, 0x1d6c0, 1},
4819         {0x1d6c2, 0x1d6da, 1},
4820         {0x1d6dc, 0x1d6fa, 1},
4821         {0x1d6fc, 0x1d714, 1},
4822         {0x1d716, 0x1d734, 1},
4823         {0x1d736, 0x1d74e, 1},
4824         {0x1d750, 0x1d76e, 1},
4825         {0x1d770, 0x1d788, 1},
4826         {0x1d78a, 0x1d7a8, 1},
4827         {0x1d7aa, 0x1d7c2, 1},
4828         {0x1d7c4, 0x1d7cb, 1},
4829         {0x1d7ce, 0x1d7ff, 1},
4830     },

```

```

4831 }
4832
4833 var _Other_Uppercase = &RangeTable{
4834     R16: []Range16{
4835         {0x2160, 0x216f, 1},
4836         {0x24b6, 0x24cf, 1},
4837     },
4838 }
4839
4840 var _Pattern_Syntax = &RangeTable{
4841     R16: []Range16{
4842         {0x0021, 0x002f, 1},
4843         {0x003a, 0x0040, 1},
4844         {0x005b, 0x005e, 1},
4845         {0x0060, 0x0060, 1},
4846         {0x007b, 0x007e, 1},
4847         {0x00a1, 0x00a7, 1},
4848         {0x00a9, 0x00a9, 1},
4849         {0x00ab, 0x00ac, 1},
4850         {0x00ae, 0x00ae, 1},
4851         {0x00b0, 0x00b1, 1},
4852         {0x00b6, 0x00b6, 1},
4853         {0x00bb, 0x00bb, 1},
4854         {0x00bf, 0x00bf, 1},
4855         {0x00d7, 0x00d7, 1},
4856         {0x00f7, 0x00f7, 1},
4857         {0x2010, 0x2027, 1},
4858         {0x2030, 0x203e, 1},
4859         {0x2041, 0x2053, 1},
4860         {0x2055, 0x205e, 1},
4861         {0x2190, 0x245f, 1},
4862         {0x2500, 0x2775, 1},
4863         {0x2794, 0x2bff, 1},
4864         {0x2e00, 0x2e7f, 1},
4865         {0x3001, 0x3003, 1},
4866         {0x3008, 0x3020, 1},
4867         {0x3030, 0x3030, 1},
4868         {0xfd3e, 0xfd3f, 1},
4869         {0xfe45, 0xfe46, 1},
4870     },
4871 }
4872
4873 var _Pattern_White_Space = &RangeTable{
4874     R16: []Range16{
4875         {0x0009, 0x000d, 1},
4876         {0x0020, 0x0020, 1},
4877         {0x0085, 0x0085, 1},
4878         {0x200e, 0x200f, 1},
4879         {0x2028, 0x2029, 1},

```

```

4880     },
4881 }
4882
4883 var _Quotation_Mark = &RangeTable{
4884     R16: []Range16{
4885         {0x0022, 0x0022, 1},
4886         {0x0027, 0x0027, 1},
4887         {0x00ab, 0x00ab, 1},
4888         {0x00bb, 0x00bb, 1},
4889         {0x2018, 0x201f, 1},
4890         {0x2039, 0x203a, 1},
4891         {0x300c, 0x300f, 1},
4892         {0x301d, 0x301f, 1},
4893         {0xfe41, 0xfe44, 1},
4894         {0xff02, 0xff02, 1},
4895         {0xff07, 0xff07, 1},
4896         {0xff62, 0xff63, 1},
4897     },
4898 }
4899
4900 var _Radical = &RangeTable{
4901     R16: []Range16{
4902         {0x2e80, 0x2e99, 1},
4903         {0x2e9b, 0x2ef3, 1},
4904         {0x2f00, 0x2fd5, 1},
4905     },
4906 }
4907
4908 var _STerm = &RangeTable{
4909     R16: []Range16{
4910         {0x0021, 0x0021, 1},
4911         {0x002e, 0x002e, 1},
4912         {0x003f, 0x003f, 1},
4913         {0x055c, 0x055c, 1},
4914         {0x055e, 0x055e, 1},
4915         {0x0589, 0x0589, 1},
4916         {0x061f, 0x061f, 1},
4917         {0x06d4, 0x06d4, 1},
4918         {0x0700, 0x0702, 1},
4919         {0x07f9, 0x07f9, 1},
4920         {0x0964, 0x0965, 1},
4921         {0x104a, 0x104b, 1},
4922         {0x1362, 0x1362, 1},
4923         {0x1367, 0x1368, 1},
4924         {0x166e, 0x166e, 1},
4925         {0x1735, 0x1736, 1},
4926         {0x1803, 0x1803, 1},
4927         {0x1809, 0x1809, 1},
4928         {0x1944, 0x1945, 1},

```

```

4929         {0x1aa8, 0x1aab, 1},
4930         {0x1b5a, 0x1b5b, 1},
4931         {0x1b5e, 0x1b5f, 1},
4932         {0x1c3b, 0x1c3c, 1},
4933         {0x1c7e, 0x1c7f, 1},
4934         {0x203c, 0x203d, 1},
4935         {0x2047, 0x2049, 1},
4936         {0x2e2e, 0x2e2e, 1},
4937         {0x3002, 0x3002, 1},
4938         {0xa4ff, 0xa4ff, 1},
4939         {0xa60e, 0xa60f, 1},
4940         {0xa6f3, 0xa6f3, 1},
4941         {0xa6f7, 0xa6f7, 1},
4942         {0xa876, 0xa877, 1},
4943         {0xa8ce, 0xa8cf, 1},
4944         {0xa92f, 0xa92f, 1},
4945         {0xa9c8, 0xa9c9, 1},
4946         {0xaa5d, 0xaa5f, 1},
4947         {0xabeb, 0xabeb, 1},
4948         {0xfe52, 0xfe52, 1},
4949         {0xfe56, 0xfe57, 1},
4950         {0xff01, 0xff01, 1},
4951         {0xff0e, 0xff0e, 1},
4952         {0xff1f, 0xff1f, 1},
4953         {0xff61, 0xff61, 1},
4954     },
4955     R32: []Range32{
4956         {0x10a56, 0x10a57, 1},
4957         {0x11047, 0x11048, 1},
4958         {0x110be, 0x110c1, 1},
4959     },
4960 }
4961
4962 var _Soft_Dotted = &RangeTable{
4963     R16: []Range16{
4964         {0x0069, 0x006a, 1},
4965         {0x012f, 0x012f, 1},
4966         {0x0249, 0x0249, 1},
4967         {0x0268, 0x0268, 1},
4968         {0x029d, 0x029d, 1},
4969         {0x02b2, 0x02b2, 1},
4970         {0x03f3, 0x03f3, 1},
4971         {0x0456, 0x0456, 1},
4972         {0x0458, 0x0458, 1},
4973         {0x1d62, 0x1d62, 1},
4974         {0x1d96, 0x1d96, 1},
4975         {0x1da4, 0x1da4, 1},
4976         {0x1da8, 0x1da8, 1},
4977         {0x1e2d, 0x1e2d, 1},
4978         {0x1ecb, 0x1ecb, 1},

```

```

4979             {0x2071, 0x2071, 1},
4980             {0x2148, 0x2149, 1},
4981             {0x2c7c, 0x2c7c, 1},
4982     },
4983     R32: []Range32{
4984             {0x1d422, 0x1d423, 1},
4985             {0x1d456, 0x1d457, 1},
4986             {0x1d48a, 0x1d48b, 1},
4987             {0x1d4be, 0x1d4bf, 1},
4988             {0x1d4f2, 0x1d4f3, 1},
4989             {0x1d526, 0x1d527, 1},
4990             {0x1d55a, 0x1d55b, 1},
4991             {0x1d58e, 0x1d58f, 1},
4992             {0x1d5c2, 0x1d5c3, 1},
4993             {0x1d5f6, 0x1d5f7, 1},
4994             {0x1d62a, 0x1d62b, 1},
4995             {0x1d65e, 0x1d65f, 1},
4996             {0x1d692, 0x1d693, 1},
4997     },
4998 }
4999
5000 var _Terminal_Punctuation = &RangeTable{
5001     R16: []Range16{
5002             {0x0021, 0x0021, 1},
5003             {0x002c, 0x002c, 1},
5004             {0x002e, 0x002e, 1},
5005             {0x003a, 0x003b, 1},
5006             {0x003f, 0x003f, 1},
5007             {0x037e, 0x037e, 1},
5008             {0x0387, 0x0387, 1},
5009             {0x0589, 0x0589, 1},
5010             {0x05c3, 0x05c3, 1},
5011             {0x060c, 0x060c, 1},
5012             {0x061b, 0x061b, 1},
5013             {0x061f, 0x061f, 1},
5014             {0x06d4, 0x06d4, 1},
5015             {0x0700, 0x070a, 1},
5016             {0x070c, 0x070c, 1},
5017             {0x07f8, 0x07f9, 1},
5018             {0x0830, 0x083e, 1},
5019             {0x085e, 0x085e, 1},
5020             {0x0964, 0x0965, 1},
5021             {0x0e5a, 0x0e5b, 1},
5022             {0x0f08, 0x0f08, 1},
5023             {0x0f0d, 0x0f12, 1},
5024             {0x104a, 0x104b, 1},
5025             {0x1361, 0x1368, 1},
5026             {0x166d, 0x166e, 1},
5027             {0x16eb, 0x16ed, 1},

```

```

5028         {0x17d4, 0x17d6, 1},
5029         {0x17da, 0x17da, 1},
5030         {0x1802, 0x1805, 1},
5031         {0x1808, 0x1809, 1},
5032         {0x1944, 0x1945, 1},
5033         {0x1aa8, 0x1aab, 1},
5034         {0x1b5a, 0x1b5b, 1},
5035         {0x1b5d, 0x1b5f, 1},
5036         {0x1c3b, 0x1c3f, 1},
5037         {0x1c7e, 0x1c7f, 1},
5038         {0x203c, 0x203d, 1},
5039         {0x2047, 0x2049, 1},
5040         {0x2e2e, 0x2e2e, 1},
5041         {0x3001, 0x3002, 1},
5042         {0xa4fe, 0xa4ff, 1},
5043         {0xa60d, 0xa60f, 1},
5044         {0xa6f3, 0xa6f7, 1},
5045         {0xa876, 0xa877, 1},
5046         {0xa8ce, 0xa8cf, 1},
5047         {0xa92f, 0xa92f, 1},
5048         {0xa9c7, 0xa9c9, 1},
5049         {0xaa5d, 0xaa5f, 1},
5050         {0xaadf, 0xaadf, 1},
5051         {0xabeb, 0xabeb, 1},
5052         {0xfe50, 0xfe52, 1},
5053         {0xfe54, 0xfe57, 1},
5054         {0xff01, 0xff01, 1},
5055         {0xff0c, 0xff0c, 1},
5056         {0xff0e, 0xff0e, 1},
5057         {0xff1a, 0xff1b, 1},
5058         {0xff1f, 0xff1f, 1},
5059         {0xff61, 0xff61, 1},
5060         {0xff64, 0xff64, 1},
5061     },
5062     R32: []Range32{
5063         {0x1039f, 0x1039f, 1},
5064         {0x103d0, 0x103d0, 1},
5065         {0x10857, 0x10857, 1},
5066         {0x1091f, 0x1091f, 1},
5067         {0x10b3a, 0x10b3f, 1},
5068         {0x11047, 0x1104d, 1},
5069         {0x110be, 0x110c1, 1},
5070         {0x12470, 0x12473, 1},
5071     },
5072 }
5073
5074 var _Unified_Ideograph = &RangeTable{
5075     R16: []Range16{
5076         {0x3400, 0x4db5, 1},

```

```

5077             {0x4e00, 0x9fcb, 1},
5078             {0xfa0e, 0xfa0f, 1},
5079             {0xfa11, 0xfa11, 1},
5080             {0xfa13, 0xfa14, 1},
5081             {0xfa1f, 0xfa1f, 1},
5082             {0xfa21, 0xfa21, 1},
5083             {0xfa23, 0xfa24, 1},
5084             {0xfa27, 0xfa29, 1},
5085         },
5086         R32: []Range32{
5087             {0x20000, 0x2a6d6, 1},
5088             {0x2a700, 0x2b734, 1},
5089             {0x2b740, 0x2b81d, 1},
5090         },
5091     }
5092
5093     var _Variation_Selector = &RangeTable{
5094         R16: []Range16{
5095             {0x180b, 0x180d, 1},
5096             {0xfe00, 0xfe0f, 1},
5097         },
5098         R32: []Range32{
5099             {0xe0100, 0xe01ef, 1},
5100         },
5101     }
5102
5103     var _White_Space = &RangeTable{
5104         R16: []Range16{
5105             {0x0009, 0x000d, 1},
5106             {0x0020, 0x0020, 1},
5107             {0x0085, 0x0085, 1},
5108             {0x00a0, 0x00a0, 1},
5109             {0x1680, 0x1680, 1},
5110             {0x180e, 0x180e, 1},
5111             {0x2000, 0x200a, 1},
5112             {0x2028, 0x2029, 1},
5113             {0x202f, 0x202f, 1},
5114             {0x205f, 0x205f, 1},
5115             {0x3000, 0x3000, 1},
5116         },
5117     }
5118
5119     // The following variables are of type *RangeTable:
5120     var (
5121         ASCII_Hex_Digit           = _ASCII_Hex_Digi
5122         Bidi_Control              = _Bidi_Control
5123         Dash                      = _Dash
5124         Deprecated               = _Deprecated
5125         Diacritic                 = _Diacritic
5126         Extender                  = _Extender

```

```

5127         Hex_Digit           = _Hex_Digit
5128         Hyphen              = _Hyphen
5129         IDS_Binary_Operator = _IDS_Binary_Ope
5130         IDS_Tertiary_Operator = _IDS_Tertiary_Op
5131         Ideographic         = _Ideographic
5132         Join_Control        = _Join_Control
5133         Logical_Order_Exception = _Logical_Order_
5134         Noncharacter_Code_Point = _Noncharacter_C
5135         Other_Alphabetic    = _Other_Alphabet
5136         Other_Default_Ignorable_Code_Point = _Other_Default_
5137         Other_Grapheme_Extend = _Other_Grapheme
5138         Other_ID_Continue   = _Other_ID_Conti
5139         Other_ID_Start      = _Other_ID_Start
5140         Other_Lowercase     = _Other_Lowercas
5141         Other_Math          = _Other_Math
5142         Other_Uppercase     = _Other_Uppercas
5143         Pattern_Syntax      = _Pattern_Syntax
5144         Pattern_White_Space = _Pattern_White_
5145         Quotation_Mark      = _Quotation_Mark
5146         Radical             = _Radical
5147         STerm               = _STerm
5148         Soft_Dotted         = _Soft_Dotted
5149         Terminal_Punctuation = _Terminal_Punct
5150         Unified_Ideograph   = _Unified_Ideogr
5151         Variation_Selector  = _Variation_Sele
5152         White_Space         = _White_Space
5153     )
5154
5155     // Generated by running
5156     //     maketables --data=http://www.unicode.org/Public/6.0.
5157     // DO NOT EDIT
5158
5159     // CaseRanges is the table describing case mappings for all
5160     // non-self mappings.
5161     var CaseRanges = _CaseRanges
5162     var _CaseRanges = []CaseRange{
5163         {0x0041, 0x005A, d{0, 32, 0}},
5164         {0x0061, 0x007A, d{-32, 0, -32}},
5165         {0x00B5, 0x00B5, d{743, 0, 743}},
5166         {0x00C0, 0x00D6, d{0, 32, 0}},
5167         {0x00D8, 0x00DE, d{0, 32, 0}},
5168         {0x00E0, 0x00F6, d{-32, 0, -32}},
5169         {0x00F8, 0x00FE, d{-32, 0, -32}},
5170         {0x00FF, 0x00FF, d{121, 0, 121}},
5171         {0x0100, 0x012F, d{UpperLower, UpperLower, UpperLowe
5172         {0x0130, 0x0130, d{0, -199, 0}},
5173         {0x0131, 0x0131, d{-232, 0, -232}},
5174         {0x0132, 0x0137, d{UpperLower, UpperLower, UpperLowe
5175         {0x0139, 0x0148, d{UpperLower, UpperLower, UpperLowe

```

5176 {0x014A, 0x0177, d{UpperLower, UpperLower, UpperLow
5177 {0x0178, 0x0178, d{0, -121, 0}},
5178 {0x0179, 0x017E, d{UpperLower, UpperLower, UpperLow
5179 {0x017F, 0x017F, d{-300, 0, -300}},
5180 {0x0180, 0x0180, d{195, 0, 195}},
5181 {0x0181, 0x0181, d{0, 210, 0}},
5182 {0x0182, 0x0185, d{UpperLower, UpperLower, UpperLow
5183 {0x0186, 0x0186, d{0, 206, 0}},
5184 {0x0187, 0x0188, d{UpperLower, UpperLower, UpperLow
5185 {0x0189, 0x018A, d{0, 205, 0}},
5186 {0x018B, 0x018C, d{UpperLower, UpperLower, UpperLow
5187 {0x018E, 0x018E, d{0, 79, 0}},
5188 {0x018F, 0x018F, d{0, 202, 0}},
5189 {0x0190, 0x0190, d{0, 203, 0}},
5190 {0x0191, 0x0192, d{UpperLower, UpperLower, UpperLow
5191 {0x0193, 0x0193, d{0, 205, 0}},
5192 {0x0194, 0x0194, d{0, 207, 0}},
5193 {0x0195, 0x0195, d{97, 0, 97}},
5194 {0x0196, 0x0196, d{0, 211, 0}},
5195 {0x0197, 0x0197, d{0, 209, 0}},
5196 {0x0198, 0x0199, d{UpperLower, UpperLower, UpperLow
5197 {0x019A, 0x019A, d{163, 0, 163}},
5198 {0x019C, 0x019C, d{0, 211, 0}},
5199 {0x019D, 0x019D, d{0, 213, 0}},
5200 {0x019E, 0x019E, d{130, 0, 130}},
5201 {0x019F, 0x019F, d{0, 214, 0}},
5202 {0x01A0, 0x01A5, d{UpperLower, UpperLower, UpperLow
5203 {0x01A6, 0x01A6, d{0, 218, 0}},
5204 {0x01A7, 0x01A8, d{UpperLower, UpperLower, UpperLow
5205 {0x01A9, 0x01A9, d{0, 218, 0}},
5206 {0x01AC, 0x01AD, d{UpperLower, UpperLower, UpperLow
5207 {0x01AE, 0x01AE, d{0, 218, 0}},
5208 {0x01AF, 0x01B0, d{UpperLower, UpperLower, UpperLow
5209 {0x01B1, 0x01B2, d{0, 217, 0}},
5210 {0x01B3, 0x01B6, d{UpperLower, UpperLower, UpperLow
5211 {0x01B7, 0x01B7, d{0, 219, 0}},
5212 {0x01B8, 0x01B9, d{UpperLower, UpperLower, UpperLow
5213 {0x01BC, 0x01BD, d{UpperLower, UpperLower, UpperLow
5214 {0x01BF, 0x01BF, d{56, 0, 56}},
5215 {0x01C4, 0x01C4, d{0, 2, 1}},
5216 {0x01C5, 0x01C5, d{-1, 1, 0}},
5217 {0x01C6, 0x01C6, d{-2, 0, -1}},
5218 {0x01C7, 0x01C7, d{0, 2, 1}},
5219 {0x01C8, 0x01C8, d{-1, 1, 0}},
5220 {0x01C9, 0x01C9, d{-2, 0, -1}},
5221 {0x01CA, 0x01CA, d{0, 2, 1}},
5222 {0x01CB, 0x01CB, d{-1, 1, 0}},
5223 {0x01CC, 0x01CC, d{-2, 0, -1}},
5224 {0x01CD, 0x01DC, d{UpperLower, UpperLower, UpperLow

5225 {0x01DD, 0x01DD, d{-79, 0, -79}},
5226 {0x01DE, 0x01EF, d{UpperLower, UpperLower, UpperLow
5227 {0x01F1, 0x01F1, d{0, 2, 1}},
5228 {0x01F2, 0x01F2, d{-1, 1, 0}},
5229 {0x01F3, 0x01F3, d{-2, 0, -1}},
5230 {0x01F4, 0x01F5, d{UpperLower, UpperLower, UpperLow
5231 {0x01F6, 0x01F6, d{0, -97, 0}},
5232 {0x01F7, 0x01F7, d{0, -56, 0}},
5233 {0x01F8, 0x021F, d{UpperLower, UpperLower, UpperLow
5234 {0x0220, 0x0220, d{0, -130, 0}},
5235 {0x0222, 0x0233, d{UpperLower, UpperLower, UpperLow
5236 {0x023A, 0x023A, d{0, 10795, 0}},
5237 {0x023B, 0x023C, d{UpperLower, UpperLower, UpperLow
5238 {0x023D, 0x023D, d{0, -163, 0}},
5239 {0x023E, 0x023E, d{0, 10792, 0}},
5240 {0x023F, 0x0240, d{10815, 0, 10815}},
5241 {0x0241, 0x0242, d{UpperLower, UpperLower, UpperLow
5242 {0x0243, 0x0243, d{0, -195, 0}},
5243 {0x0244, 0x0244, d{0, 69, 0}},
5244 {0x0245, 0x0245, d{0, 71, 0}},
5245 {0x0246, 0x024F, d{UpperLower, UpperLower, UpperLow
5246 {0x0250, 0x0250, d{10783, 0, 10783}},
5247 {0x0251, 0x0251, d{10780, 0, 10780}},
5248 {0x0252, 0x0252, d{10782, 0, 10782}},
5249 {0x0253, 0x0253, d{-210, 0, -210}},
5250 {0x0254, 0x0254, d{-206, 0, -206}},
5251 {0x0256, 0x0257, d{-205, 0, -205}},
5252 {0x0259, 0x0259, d{-202, 0, -202}},
5253 {0x025B, 0x025B, d{-203, 0, -203}},
5254 {0x0260, 0x0260, d{-205, 0, -205}},
5255 {0x0263, 0x0263, d{-207, 0, -207}},
5256 {0x0265, 0x0265, d{42280, 0, 42280}},
5257 {0x0268, 0x0268, d{-209, 0, -209}},
5258 {0x0269, 0x0269, d{-211, 0, -211}},
5259 {0x026B, 0x026B, d{10743, 0, 10743}},
5260 {0x026F, 0x026F, d{-211, 0, -211}},
5261 {0x0271, 0x0271, d{10749, 0, 10749}},
5262 {0x0272, 0x0272, d{-213, 0, -213}},
5263 {0x0275, 0x0275, d{-214, 0, -214}},
5264 {0x027D, 0x027D, d{10727, 0, 10727}},
5265 {0x0280, 0x0280, d{-218, 0, -218}},
5266 {0x0283, 0x0283, d{-218, 0, -218}},
5267 {0x0288, 0x0288, d{-218, 0, -218}},
5268 {0x0289, 0x0289, d{-69, 0, -69}},
5269 {0x028A, 0x028B, d{-217, 0, -217}},
5270 {0x028C, 0x028C, d{-71, 0, -71}},
5271 {0x0292, 0x0292, d{-219, 0, -219}},
5272 {0x0345, 0x0345, d{84, 0, 84}},
5273 {0x0370, 0x0373, d{UpperLower, UpperLower, UpperLow
5274 {0x0376, 0x0377, d{UpperLower, UpperLower, UpperLow

5275 {0x037B, 0x037D, d{130, 0, 130}},
5276 {0x0386, 0x0386, d{0, 38, 0}},
5277 {0x0388, 0x038A, d{0, 37, 0}},
5278 {0x038C, 0x038C, d{0, 64, 0}},
5279 {0x038E, 0x038F, d{0, 63, 0}},
5280 {0x0391, 0x03A1, d{0, 32, 0}},
5281 {0x03A3, 0x03AB, d{0, 32, 0}},
5282 {0x03AC, 0x03AC, d{-38, 0, -38}},
5283 {0x03AD, 0x03AF, d{-37, 0, -37}},
5284 {0x03B1, 0x03C1, d{-32, 0, -32}},
5285 {0x03C2, 0x03C2, d{-31, 0, -31}},
5286 {0x03C3, 0x03CB, d{-32, 0, -32}},
5287 {0x03CC, 0x03CC, d{-64, 0, -64}},
5288 {0x03CD, 0x03CE, d{-63, 0, -63}},
5289 {0x03CF, 0x03CF, d{0, 8, 0}},
5290 {0x03D0, 0x03D0, d{-62, 0, -62}},
5291 {0x03D1, 0x03D1, d{-57, 0, -57}},
5292 {0x03D5, 0x03D5, d{-47, 0, -47}},
5293 {0x03D6, 0x03D6, d{-54, 0, -54}},
5294 {0x03D7, 0x03D7, d{-8, 0, -8}},
5295 {0x03D8, 0x03EF, d{UpperLower, UpperLower, UpperLow
5296 {0x03F0, 0x03F0, d{-86, 0, -86}},
5297 {0x03F1, 0x03F1, d{-80, 0, -80}},
5298 {0x03F2, 0x03F2, d{7, 0, 7}},
5299 {0x03F4, 0x03F4, d{0, -60, 0}},
5300 {0x03F5, 0x03F5, d{-96, 0, -96}},
5301 {0x03F7, 0x03F8, d{UpperLower, UpperLower, UpperLow
5302 {0x03F9, 0x03F9, d{0, -7, 0}},
5303 {0x03FA, 0x03FB, d{UpperLower, UpperLower, UpperLow
5304 {0x03FD, 0x03FF, d{0, -130, 0}},
5305 {0x0400, 0x040F, d{0, 80, 0}},
5306 {0x0410, 0x042F, d{0, 32, 0}},
5307 {0x0430, 0x044F, d{-32, 0, -32}},
5308 {0x0450, 0x045F, d{-80, 0, -80}},
5309 {0x0460, 0x0481, d{UpperLower, UpperLower, UpperLow
5310 {0x048A, 0x04BF, d{UpperLower, UpperLower, UpperLow
5311 {0x04C0, 0x04C0, d{0, 15, 0}},
5312 {0x04C1, 0x04CE, d{UpperLower, UpperLower, UpperLow
5313 {0x04CF, 0x04CF, d{-15, 0, -15}},
5314 {0x04D0, 0x0527, d{UpperLower, UpperLower, UpperLow
5315 {0x0531, 0x0556, d{0, 48, 0}},
5316 {0x0561, 0x0586, d{-48, 0, -48}},
5317 {0x10A0, 0x10C5, d{0, 7264, 0}},
5318 {0x1D79, 0x1D79, d{35332, 0, 35332}},
5319 {0x1D7D, 0x1D7D, d{3814, 0, 3814}},
5320 {0x1E00, 0x1E95, d{UpperLower, UpperLower, UpperLow
5321 {0x1E9B, 0x1E9B, d{-59, 0, -59}},
5322 {0x1E9E, 0x1E9E, d{0, -7615, 0}},
5323 {0x1EA0, 0x1EFF, d{UpperLower, UpperLower, UpperLow

5324 {0x1F00, 0x1F07, d{8, 0, 8}},
5325 {0x1F08, 0x1F0F, d{0, -8, 0}},
5326 {0x1F10, 0x1F15, d{8, 0, 8}},
5327 {0x1F18, 0x1F1D, d{0, -8, 0}},
5328 {0x1F20, 0x1F27, d{8, 0, 8}},
5329 {0x1F28, 0x1F2F, d{0, -8, 0}},
5330 {0x1F30, 0x1F37, d{8, 0, 8}},
5331 {0x1F38, 0x1F3F, d{0, -8, 0}},
5332 {0x1F40, 0x1F45, d{8, 0, 8}},
5333 {0x1F48, 0x1F4D, d{0, -8, 0}},
5334 {0x1F51, 0x1F51, d{8, 0, 8}},
5335 {0x1F53, 0x1F53, d{8, 0, 8}},
5336 {0x1F55, 0x1F55, d{8, 0, 8}},
5337 {0x1F57, 0x1F57, d{8, 0, 8}},
5338 {0x1F59, 0x1F59, d{0, -8, 0}},
5339 {0x1F5B, 0x1F5B, d{0, -8, 0}},
5340 {0x1F5D, 0x1F5D, d{0, -8, 0}},
5341 {0x1F5F, 0x1F5F, d{0, -8, 0}},
5342 {0x1F60, 0x1F67, d{8, 0, 8}},
5343 {0x1F68, 0x1F6F, d{0, -8, 0}},
5344 {0x1F70, 0x1F71, d{74, 0, 74}},
5345 {0x1F72, 0x1F75, d{86, 0, 86}},
5346 {0x1F76, 0x1F77, d{100, 0, 100}},
5347 {0x1F78, 0x1F79, d{128, 0, 128}},
5348 {0x1F7A, 0x1F7B, d{112, 0, 112}},
5349 {0x1F7C, 0x1F7D, d{126, 0, 126}},
5350 {0x1F80, 0x1F87, d{8, 0, 8}},
5351 {0x1F88, 0x1F8F, d{0, -8, 0}},
5352 {0x1F90, 0x1F97, d{8, 0, 8}},
5353 {0x1F98, 0x1F9F, d{0, -8, 0}},
5354 {0x1FA0, 0x1FA7, d{8, 0, 8}},
5355 {0x1FA8, 0x1FAF, d{0, -8, 0}},
5356 {0x1FB0, 0x1FB1, d{8, 0, 8}},
5357 {0x1FB3, 0x1FB3, d{9, 0, 9}},
5358 {0x1FB8, 0x1FB9, d{0, -8, 0}},
5359 {0x1FBA, 0x1FBB, d{0, -74, 0}},
5360 {0x1FBC, 0x1FBC, d{0, -9, 0}},
5361 {0x1FBE, 0x1FBE, d{-7205, 0, -7205}},
5362 {0x1FC3, 0x1FC3, d{9, 0, 9}},
5363 {0x1FC8, 0x1FCB, d{0, -86, 0}},
5364 {0x1FCC, 0x1FCC, d{0, -9, 0}},
5365 {0x1FD0, 0x1FD1, d{8, 0, 8}},
5366 {0x1FD8, 0x1FD9, d{0, -8, 0}},
5367 {0x1FDA, 0x1FDB, d{0, -100, 0}},
5368 {0x1FE0, 0x1FE1, d{8, 0, 8}},
5369 {0x1FE5, 0x1FE5, d{7, 0, 7}},
5370 {0x1FE8, 0x1FE9, d{0, -8, 0}},
5371 {0x1FEA, 0x1FEB, d{0, -112, 0}},
5372 {0x1FEC, 0x1FEC, d{0, -7, 0}},

```

5373     {0x1FF3, 0x1FF3, d{9, 0, 9}},
5374     {0x1FF8, 0x1FF9, d{0, -128, 0}},
5375     {0x1FFA, 0x1FFB, d{0, -126, 0}},
5376     {0x1FFC, 0x1FFC, d{0, -9, 0}},
5377     {0x2126, 0x2126, d{0, -7517, 0}},
5378     {0x212A, 0x212A, d{0, -8383, 0}},
5379     {0x212B, 0x212B, d{0, -8262, 0}},
5380     {0x2132, 0x2132, d{0, 28, 0}},
5381     {0x214E, 0x214E, d{-28, 0, -28}},
5382     {0x2160, 0x216F, d{0, 16, 0}},
5383     {0x2170, 0x217F, d{-16, 0, -16}},
5384     {0x2183, 0x2184, d{UpperLower, UpperLower, UpperLowe
5385     {0x24B6, 0x24CF, d{0, 26, 0}},
5386     {0x24D0, 0x24E9, d{-26, 0, -26}},
5387     {0x2C00, 0x2C2E, d{0, 48, 0}},
5388     {0x2C30, 0x2C5E, d{-48, 0, -48}},
5389     {0x2C60, 0x2C61, d{UpperLower, UpperLower, UpperLowe
5390     {0x2C62, 0x2C62, d{0, -10743, 0}},
5391     {0x2C63, 0x2C63, d{0, -3814, 0}},
5392     {0x2C64, 0x2C64, d{0, -10727, 0}},
5393     {0x2C65, 0x2C65, d{-10795, 0, -10795}},
5394     {0x2C66, 0x2C66, d{-10792, 0, -10792}},
5395     {0x2C67, 0x2C6C, d{UpperLower, UpperLower, UpperLowe
5396     {0x2C6D, 0x2C6D, d{0, -10780, 0}},
5397     {0x2C6E, 0x2C6E, d{0, -10749, 0}},
5398     {0x2C6F, 0x2C6F, d{0, -10783, 0}},
5399     {0x2C70, 0x2C70, d{0, -10782, 0}},
5400     {0x2C72, 0x2C73, d{UpperLower, UpperLower, UpperLowe
5401     {0x2C75, 0x2C76, d{UpperLower, UpperLower, UpperLowe
5402     {0x2C7E, 0x2C7F, d{0, -10815, 0}},
5403     {0x2C80, 0x2CE3, d{UpperLower, UpperLower, UpperLowe
5404     {0x2CEB, 0x2CEE, d{UpperLower, UpperLower, UpperLowe
5405     {0x2D00, 0x2D25, d{-7264, 0, -7264}},
5406     {0xA640, 0xA66D, d{UpperLower, UpperLower, UpperLowe
5407     {0xA680, 0xA697, d{UpperLower, UpperLower, UpperLowe
5408     {0xA722, 0xA72F, d{UpperLower, UpperLower, UpperLowe
5409     {0xA732, 0xA76F, d{UpperLower, UpperLower, UpperLowe
5410     {0xA779, 0xA77C, d{UpperLower, UpperLower, UpperLowe
5411     {0xA77D, 0xA77D, d{0, -35332, 0}},
5412     {0xA77E, 0xA787, d{UpperLower, UpperLower, UpperLowe
5413     {0xA78B, 0xA78C, d{UpperLower, UpperLower, UpperLowe
5414     {0xA78D, 0xA78D, d{0, -42280, 0}},
5415     {0xA790, 0xA791, d{UpperLower, UpperLower, UpperLowe
5416     {0xA7A0, 0xA7A9, d{UpperLower, UpperLower, UpperLowe
5417     {0xFF21, 0xFF3A, d{0, 32, 0}},
5418     {0xFF41, 0xFF5A, d{-32, 0, -32}},
5419     {0x10400, 0x10427, d{0, 40, 0}},
5420     {0x10428, 0x1044F, d{-40, 0, -40}},
5421 }
5422 var properties = [MaxLatin1 + 1]uint8{

```

```

5423      0x00: pC,      // '\x00'
5424      0x01: pC,      // '\x01'
5425      0x02: pC,      // '\x02'
5426      0x03: pC,      // '\x03'
5427      0x04: pC,      // '\x04'
5428      0x05: pC,      // '\x05'
5429      0x06: pC,      // '\x06'
5430      0x07: pC,      // '\a'
5431      0x08: pC,      // '\b'
5432      0x09: pC,      // '\t'
5433      0x0A: pC,      // '\n'
5434      0x0B: pC,      // '\v'
5435      0x0C: pC,      // '\f'
5436      0x0D: pC,      // '\r'
5437      0x0E: pC,      // '\x0e'
5438      0x0F: pC,      // '\x0f'
5439      0x10: pC,      // '\x10'
5440      0x11: pC,      // '\x11'
5441      0x12: pC,      // '\x12'
5442      0x13: pC,      // '\x13'
5443      0x14: pC,      // '\x14'
5444      0x15: pC,      // '\x15'
5445      0x16: pC,      // '\x16'
5446      0x17: pC,      // '\x17'
5447      0x18: pC,      // '\x18'
5448      0x19: pC,      // '\x19'
5449      0x1A: pC,      // '\x1a'
5450      0x1B: pC,      // '\x1b'
5451      0x1C: pC,      // '\x1c'
5452      0x1D: pC,      // '\x1d'
5453      0x1E: pC,      // '\x1e'
5454      0x1F: pC,      // '\x1f'
5455      0x20: pZ | pp,  // ' '
5456      0x21: pP | pp,  // '!'
5457      0x22: pP | pp,  // '"'
5458      0x23: pP | pp,  // '#'
5459      0x24: pS | pp,  // '$'
5460      0x25: pP | pp,  // '%'
5461      0x26: pP | pp,  // '&'
5462      0x27: pP | pp,  // '\ '
5463      0x28: pP | pp,  // '('
5464      0x29: pP | pp,  // ')'
5465      0x2A: pP | pp,  // '*'
5466      0x2B: pS | pp,  // '+'
5467      0x2C: pP | pp,  // ','
5468      0x2D: pP | pp,  // '-'
5469      0x2E: pP | pp,  // '.'
5470      0x2F: pP | pp,  // '/'
5471      0x30: pN | pp,  // '0'

```

5472	0x31:	pN		pp,	//	'1'
5473	0x32:	pN		pp,	//	'2'
5474	0x33:	pN		pp,	//	'3'
5475	0x34:	pN		pp,	//	'4'
5476	0x35:	pN		pp,	//	'5'
5477	0x36:	pN		pp,	//	'6'
5478	0x37:	pN		pp,	//	'7'
5479	0x38:	pN		pp,	//	'8'
5480	0x39:	pN		pp,	//	'9'
5481	0x3A:	pP		pp,	//	':'
5482	0x3B:	pP		pp,	//	','
5483	0x3C:	pS		pp,	//	'<'
5484	0x3D:	pS		pp,	//	'='
5485	0x3E:	pS		pp,	//	'>'
5486	0x3F:	pP		pp,	//	'?'
5487	0x40:	pP		pp,	//	'@'
5488	0x41:	pLu		pp,	//	'A'
5489	0x42:	pLu		pp,	//	'B'
5490	0x43:	pLu		pp,	//	'C'
5491	0x44:	pLu		pp,	//	'D'
5492	0x45:	pLu		pp,	//	'E'
5493	0x46:	pLu		pp,	//	'F'
5494	0x47:	pLu		pp,	//	'G'
5495	0x48:	pLu		pp,	//	'H'
5496	0x49:	pLu		pp,	//	'I'
5497	0x4A:	pLu		pp,	//	'J'
5498	0x4B:	pLu		pp,	//	'K'
5499	0x4C:	pLu		pp,	//	'L'
5500	0x4D:	pLu		pp,	//	'M'
5501	0x4E:	pLu		pp,	//	'N'
5502	0x4F:	pLu		pp,	//	'O'
5503	0x50:	pLu		pp,	//	'P'
5504	0x51:	pLu		pp,	//	'Q'
5505	0x52:	pLu		pp,	//	'R'
5506	0x53:	pLu		pp,	//	'S'
5507	0x54:	pLu		pp,	//	'T'
5508	0x55:	pLu		pp,	//	'U'
5509	0x56:	pLu		pp,	//	'V'
5510	0x57:	pLu		pp,	//	'W'
5511	0x58:	pLu		pp,	//	'X'
5512	0x59:	pLu		pp,	//	'Y'
5513	0x5A:	pLu		pp,	//	'Z'
5514	0x5B:	pP		pp,	//	'['
5515	0x5C:	pP		pp,	//	'\'
5516	0x5D:	pP		pp,	//	']'
5517	0x5E:	pS		pp,	//	'^'
5518	0x5F:	pP		pp,	//	'_'
5519	0x60:	pS		pp,	//	'`'
5520	0x61:	pL1		pp,	//	'a'

5521	0x62:	pL1		pp,	//	'b'
5522	0x63:	pL1		pp,	//	'c'
5523	0x64:	pL1		pp,	//	'd'
5524	0x65:	pL1		pp,	//	'e'
5525	0x66:	pL1		pp,	//	'f'
5526	0x67:	pL1		pp,	//	'g'
5527	0x68:	pL1		pp,	//	'h'
5528	0x69:	pL1		pp,	//	'i'
5529	0x6A:	pL1		pp,	//	'j'
5530	0x6B:	pL1		pp,	//	'k'
5531	0x6C:	pL1		pp,	//	'l'
5532	0x6D:	pL1		pp,	//	'm'
5533	0x6E:	pL1		pp,	//	'n'
5534	0x6F:	pL1		pp,	//	'o'
5535	0x70:	pL1		pp,	//	'p'
5536	0x71:	pL1		pp,	//	'q'
5537	0x72:	pL1		pp,	//	'r'
5538	0x73:	pL1		pp,	//	's'
5539	0x74:	pL1		pp,	//	't'
5540	0x75:	pL1		pp,	//	'u'
5541	0x76:	pL1		pp,	//	'v'
5542	0x77:	pL1		pp,	//	'w'
5543	0x78:	pL1		pp,	//	'x'
5544	0x79:	pL1		pp,	//	'y'
5545	0x7A:	pL1		pp,	//	'z'
5546	0x7B:	pP		pp,	//	'{'
5547	0x7C:	pS		pp,	//	' '
5548	0x7D:	pP		pp,	//	'}'
5549	0x7E:	pS		pp,	//	'~'
5550	0x7F:	pC,			//	'\u007f'
5551	0x80:	pC,			//	'\u0080'
5552	0x81:	pC,			//	'\u0081'
5553	0x82:	pC,			//	'\u0082'
5554	0x83:	pC,			//	'\u0083'
5555	0x84:	pC,			//	'\u0084'
5556	0x85:	pC,			//	'\u0085'
5557	0x86:	pC,			//	'\u0086'
5558	0x87:	pC,			//	'\u0087'
5559	0x88:	pC,			//	'\u0088'
5560	0x89:	pC,			//	'\u0089'
5561	0x8A:	pC,			//	'\u008a'
5562	0x8B:	pC,			//	'\u008b'
5563	0x8C:	pC,			//	'\u008c'
5564	0x8D:	pC,			//	'\u008d'
5565	0x8E:	pC,			//	'\u008e'
5566	0x8F:	pC,			//	'\u008f'
5567	0x90:	pC,			//	'\u0090'
5568	0x91:	pC,			//	'\u0091'
5569	0x92:	pC,			//	'\u0092'
5570	0x93:	pC,			//	'\u0093'

```

5571      0x94: pC,      // '\u0094'
5572      0x95: pC,      // '\u0095'
5573      0x96: pC,      // '\u0096'
5574      0x97: pC,      // '\u0097'
5575      0x98: pC,      // '\u0098'
5576      0x99: pC,      // '\u0099'
5577      0x9A: pC,      // '\u009a'
5578      0x9B: pC,      // '\u009b'
5579      0x9C: pC,      // '\u009c'
5580      0x9D: pC,      // '\u009d'
5581      0x9E: pC,      // '\u009e'
5582      0x9F: pC,      // '\u009f'
5583      0xA0: pZ,      // '\u00a0'
5584      0xA1: pP | pp, // 'i'
5585      0xA2: pS | pp, // 'ç'
5586      0xA3: pS | pp, // '£'
5587      0xA4: pS | pp, // '¤'
5588      0xA5: pS | pp, // '¥'
5589      0xA6: pS | pp, // '¦'
5590      0xA7: pS | pp, // '§'
5591      0xA8: pS | pp, // '¨'
5592      0xA9: pS | pp, // '©'
5593      0xAA: pL1 | pp, // 'á'
5594      0xAB: pP | pp, // '«'
5595      0xAC: pS | pp, // '¬'
5596      0xAD: 0,      // '\u00ad'
5597      0xAE: pS | pp, // '®'
5598      0xAF: pS | pp, // '¯'
5599      0xB0: pS | pp, // '°'
5600      0xB1: pS | pp, // '±'
5601      0xB2: pN | pp, // '²'
5602      0xB3: pN | pp, // '³'
5603      0xB4: pS | pp, // '´'
5604      0xB5: pL1 | pp, // 'µ'
5605      0xB6: pS | pp, // '¶'
5606      0xB7: pP | pp, // '·'
5607      0xB8: pS | pp, // '¸'
5608      0xB9: pN | pp, // '¹'
5609      0xBA: pL1 | pp, // 'º'
5610      0xBB: pP | pp, // '»'
5611      0xBC: pN | pp, // '¼'
5612      0xBD: pN | pp, // '½'
5613      0xBE: pN | pp, // '¾'
5614      0xBF: pP | pp, // '¿'
5615      0xC0: pLu | pp, // 'À'
5616      0xC1: pLu | pp, // 'Á'
5617      0xC2: pLu | pp, // 'Â'
5618      0xC3: pLu | pp, // 'Ã'
5619      0xC4: pLu | pp, // 'Ä'

```

5620	0xC5:	pLu		pp,	//	'Å'
5621	0xC6:	pLu		pp,	//	'Æ'
5622	0xC7:	pLu		pp,	//	'Ç'
5623	0xC8:	pLu		pp,	//	'È'
5624	0xC9:	pLu		pp,	//	'É'
5625	0xCA:	pLu		pp,	//	'Ê'
5626	0xCB:	pLu		pp,	//	'Ë'
5627	0xCC:	pLu		pp,	//	'Ì'
5628	0xCD:	pLu		pp,	//	'Í'
5629	0xCE:	pLu		pp,	//	'Î'
5630	0xCF:	pLu		pp,	//	'Ï'
5631	0xD0:	pLu		pp,	//	'Ð'
5632	0xD1:	pLu		pp,	//	'Ñ'
5633	0xD2:	pLu		pp,	//	'Ò'
5634	0xD3:	pLu		pp,	//	'Ó'
5635	0xD4:	pLu		pp,	//	'Ô'
5636	0xD5:	pLu		pp,	//	'Õ'
5637	0xD6:	pLu		pp,	//	'Ö'
5638	0xD7:	pS		pp,	//	'×
5639	0xD8:	pLu		pp,	//	'Ø'
5640	0xD9:	pLu		pp,	//	'Ù'
5641	0xDA:	pLu		pp,	//	'Ú'
5642	0xDB:	pLu		pp,	//	'Û'
5643	0xDC:	pLu		pp,	//	'Ü'
5644	0xDD:	pLu		pp,	//	'Ý'
5645	0xDE:	pLu		pp,	//	'Þ'
5646	0xDF:	pLl		pp,	//	'ß'
5647	0xE0:	pLl		pp,	//	'à'
5648	0xE1:	pLl		pp,	//	'á'
5649	0xE2:	pLl		pp,	//	'â'
5650	0xE3:	pLl		pp,	//	'ã'
5651	0xE4:	pLl		pp,	//	'ä'
5652	0xE5:	pLl		pp,	//	'å'
5653	0xE6:	pLl		pp,	//	'æ'
5654	0xE7:	pLl		pp,	//	'ç'
5655	0xE8:	pLl		pp,	//	'è'
5656	0xE9:	pLl		pp,	//	'é'
5657	0xEA:	pLl		pp,	//	'ê'
5658	0xEB:	pLl		pp,	//	'ë'
5659	0xEC:	pLl		pp,	//	'ì'
5660	0xED:	pLl		pp,	//	'í'
5661	0xEE:	pLl		pp,	//	'î'
5662	0xEF:	pLl		pp,	//	'ï'
5663	0xF0:	pLl		pp,	//	'ð'
5664	0xF1:	pLl		pp,	//	'ñ'
5665	0xF2:	pLl		pp,	//	'ò'
5666	0xF3:	pLl		pp,	//	'ó'
5667	0xF4:	pLl		pp,	//	'ô'
5668	0xF5:	pLl		pp,	//	'õ'

```

5669         0xF6: pLl | pp, // 'ö'
5670         0xF7: pS | pp, // '÷'
5671         0xF8: pLl | pp, // 'ø'
5672         0xF9: pLl | pp, // 'ù'
5673         0xFA: pLl | pp, // 'ú'
5674         0xFB: pLl | pp, // 'û'
5675         0xFC: pLl | pp, // 'ü'
5676         0xFD: pLl | pp, // 'ý'
5677         0xFE: pLl | pp, // 'þ'
5678         0xFF: pLl | pp, // 'ÿ'
5679     }
5680
5681     var caseOrbit = []foldPair{
5682         {0x004B, 0x006B},
5683         {0x0053, 0x0073},
5684         {0x006B, 0x212A},
5685         {0x0073, 0x017F},
5686         {0x00B5, 0x039C},
5687         {0x00C5, 0x00E5},
5688         {0x00DF, 0x1E9E},
5689         {0x00E5, 0x212B},
5690         {0x0130, 0x0130},
5691         {0x0131, 0x0131},
5692         {0x017F, 0x0053},
5693         {0x01C4, 0x01C5},
5694         {0x01C5, 0x01C6},
5695         {0x01C6, 0x01C4},
5696         {0x01C7, 0x01C8},
5697         {0x01C8, 0x01C9},
5698         {0x01C9, 0x01C7},
5699         {0x01CA, 0x01CB},
5700         {0x01CB, 0x01CC},
5701         {0x01CC, 0x01CA},
5702         {0x01F1, 0x01F2},
5703         {0x01F2, 0x01F3},
5704         {0x01F3, 0x01F1},
5705         {0x0345, 0x0399},
5706         {0x0392, 0x03B2},
5707         {0x0395, 0x03B5},
5708         {0x0398, 0x03B8},
5709         {0x0399, 0x03B9},
5710         {0x039A, 0x03BA},
5711         {0x039C, 0x03BC},
5712         {0x03A0, 0x03C0},
5713         {0x03A1, 0x03C1},
5714         {0x03A3, 0x03C2},
5715         {0x03A6, 0x03C6},
5716         {0x03A9, 0x03C9},
5717         {0x03B2, 0x03D0},
5718         {0x03B5, 0x03F5},

```

```

5719         {0x03B8, 0x03D1},
5720         {0x03B9, 0x1FBE},
5721         {0x03BA, 0x03F0},
5722         {0x03BC, 0x00B5},
5723         {0x03C0, 0x03D6},
5724         {0x03C1, 0x03F1},
5725         {0x03C2, 0x03C3},
5726         {0x03C3, 0x03A3},
5727         {0x03C6, 0x03D5},
5728         {0x03C9, 0x2126},
5729         {0x03D0, 0x0392},
5730         {0x03D1, 0x03F4},
5731         {0x03D5, 0x03A6},
5732         {0x03D6, 0x03A0},
5733         {0x03F0, 0x039A},
5734         {0x03F1, 0x03A1},
5735         {0x03F4, 0x0398},
5736         {0x03F5, 0x0395},
5737         {0x1E60, 0x1E61},
5738         {0x1E61, 0x1E9B},
5739         {0x1E9B, 0x1E60},
5740         {0x1E9E, 0x00DF},
5741         {0x1FBE, 0x0345},
5742         {0x2126, 0x03A9},
5743         {0x212A, 0x004B},
5744         {0x212B, 0x00C5},
5745     }
5746
5747     // FoldCategory maps a category name to a table of
5748     // code points outside the category that are equivalent unde
5749     // simple case folding to code points inside the category.
5750     // If there is no entry for a category name, there are no su
5751     var FoldCategory = map[string]*RangeTable{
5752         "Common":    foldCommon,
5753         "Greek":      foldGreek,
5754         "Inherited":  foldInherited,
5755         "L":          foldL,
5756         "Ll":         foldLl,
5757         "Lt":         foldLt,
5758         "Lu":         foldLu,
5759         "M":          foldM,
5760         "Mn":         foldMn,
5761     }
5762
5763     var foldCommon = &RangeTable{
5764         R16: []Range16{
5765             {0x039c, 0x03bc, 32},
5766         },
5767     }

```

```

5768
5769 var foldGreek = &RangeTable{
5770     R16: []Range16{
5771         {0x00b5, 0x0345, 656},
5772     },
5773 }
5774
5775 var foldInherited = &RangeTable{
5776     R16: []Range16{
5777         {0x0399, 0x03b9, 32},
5778         {0x1fbe, 0x1fbe, 1},
5779     },
5780 }
5781
5782 var foldL = &RangeTable{
5783     R16: []Range16{
5784         {0x0345, 0x0345, 1},
5785     },
5786 }
5787
5788 var foldL1 = &RangeTable{
5789     R16: []Range16{
5790         {0x0041, 0x005a, 1},
5791         {0x00c0, 0x00d6, 1},
5792         {0x00d8, 0x00de, 1},
5793         {0x0100, 0x012e, 2},
5794         {0x0132, 0x0136, 2},
5795         {0x0139, 0x0147, 2},
5796         {0x014a, 0x0178, 2},
5797         {0x0179, 0x017d, 2},
5798         {0x0181, 0x0182, 1},
5799         {0x0184, 0x0186, 2},
5800         {0x0187, 0x0189, 2},
5801         {0x018a, 0x018b, 1},
5802         {0x018e, 0x0191, 1},
5803         {0x0193, 0x0194, 1},
5804         {0x0196, 0x0198, 1},
5805         {0x019c, 0x019d, 1},
5806         {0x019f, 0x01a0, 1},
5807         {0x01a2, 0x01a6, 2},
5808         {0x01a7, 0x01a9, 2},
5809         {0x01ac, 0x01ae, 2},
5810         {0x01af, 0x01b1, 2},
5811         {0x01b2, 0x01b3, 1},
5812         {0x01b5, 0x01b7, 2},
5813         {0x01b8, 0x01bc, 4},
5814         {0x01c4, 0x01c5, 1},
5815         {0x01c7, 0x01c8, 1},
5816         {0x01ca, 0x01cb, 1},

```

5817 {0x01cd, 0x01db, 2},
5818 {0x01de, 0x01ee, 2},
5819 {0x01f1, 0x01f2, 1},
5820 {0x01f4, 0x01f6, 2},
5821 {0x01f7, 0x01f8, 1},
5822 {0x01fa, 0x0232, 2},
5823 {0x023a, 0x023b, 1},
5824 {0x023d, 0x023e, 1},
5825 {0x0241, 0x0243, 2},
5826 {0x0244, 0x0246, 1},
5827 {0x0248, 0x024e, 2},
5828 {0x0345, 0x0370, 43},
5829 {0x0372, 0x0376, 4},
5830 {0x0386, 0x0388, 2},
5831 {0x0389, 0x038a, 1},
5832 {0x038c, 0x038e, 2},
5833 {0x038f, 0x0391, 2},
5834 {0x0392, 0x03a1, 1},
5835 {0x03a3, 0x03ab, 1},
5836 {0x03cf, 0x03d8, 9},
5837 {0x03da, 0x03ee, 2},
5838 {0x03f4, 0x03f7, 3},
5839 {0x03f9, 0x03fa, 1},
5840 {0x03fd, 0x042f, 1},
5841 {0x0460, 0x0480, 2},
5842 {0x048a, 0x04c0, 2},
5843 {0x04c1, 0x04cd, 2},
5844 {0x04d0, 0x0526, 2},
5845 {0x0531, 0x0556, 1},
5846 {0x10a0, 0x10c5, 1},
5847 {0x1e00, 0x1e94, 2},
5848 {0x1e9e, 0x1efe, 2},
5849 {0x1f08, 0x1f0f, 1},
5850 {0x1f18, 0x1f1d, 1},
5851 {0x1f28, 0x1f2f, 1},
5852 {0x1f38, 0x1f3f, 1},
5853 {0x1f48, 0x1f4d, 1},
5854 {0x1f59, 0x1f5f, 2},
5855 {0x1f68, 0x1f6f, 1},
5856 {0x1f88, 0x1f8f, 1},
5857 {0x1f98, 0x1f9f, 1},
5858 {0x1fa8, 0x1faf, 1},
5859 {0x1fb8, 0x1fbc, 1},
5860 {0x1fc8, 0x1fcc, 1},
5861 {0x1fd8, 0x1fdb, 1},
5862 {0x1fe8, 0x1fec, 1},
5863 {0x1ff8, 0x1ffc, 1},
5864 {0x2126, 0x212a, 4},
5865 {0x212b, 0x2132, 7},
5866 {0x2183, 0x2c00, 2685},

```

5867         {0x2c01, 0x2c2e, 1},
5868         {0x2c60, 0x2c62, 2},
5869         {0x2c63, 0x2c64, 1},
5870         {0x2c67, 0x2c6d, 2},
5871         {0x2c6e, 0x2c70, 1},
5872         {0x2c72, 0x2c75, 3},
5873         {0x2c7e, 0x2c80, 1},
5874         {0x2c82, 0x2ce2, 2},
5875         {0x2ceb, 0x2ced, 2},
5876         {0xa640, 0xa66c, 2},
5877         {0xa680, 0xa696, 2},
5878         {0xa722, 0xa72e, 2},
5879         {0xa732, 0xa76e, 2},
5880         {0xa779, 0xa77d, 2},
5881         {0xa77e, 0xa786, 2},
5882         {0xa78b, 0xa78d, 2},
5883         {0xa790, 0xa7a0, 16},
5884         {0xa7a2, 0xa7a8, 2},
5885         {0xff21, 0xff3a, 1},
5886     },
5887     R32: []Range32{
5888         {0x10400, 0x10427, 1},
5889     },
5890 }
5891
5892 var foldLt = &RangeTable{
5893     R16: []Range16{
5894         {0x01c4, 0x01c6, 2},
5895         {0x01c7, 0x01c9, 2},
5896         {0x01ca, 0x01cc, 2},
5897         {0x01f1, 0x01f3, 2},
5898         {0x1f80, 0x1f87, 1},
5899         {0x1f90, 0x1f97, 1},
5900         {0x1fa0, 0x1fa7, 1},
5901         {0x1fb3, 0x1fc3, 16},
5902         {0x1ff3, 0x1ff3, 1},
5903     },
5904 }
5905
5906 var foldLu = &RangeTable{
5907     R16: []Range16{
5908         {0x0061, 0x007a, 1},
5909         {0x00b5, 0x00df, 42},
5910         {0x00e0, 0x00f6, 1},
5911         {0x00f8, 0x00ff, 1},
5912         {0x0101, 0x012f, 2},
5913         {0x0133, 0x0137, 2},
5914         {0x013a, 0x0148, 2},
5915         {0x014b, 0x0177, 2},

```

5916 {0x017a, 0x017e, 2},
5917 {0x017f, 0x0180, 1},
5918 {0x0183, 0x0185, 2},
5919 {0x0188, 0x018c, 4},
5920 {0x0192, 0x0195, 3},
5921 {0x0199, 0x019a, 1},
5922 {0x019e, 0x01a1, 3},
5923 {0x01a3, 0x01a5, 2},
5924 {0x01a8, 0x01ad, 5},
5925 {0x01b0, 0x01b4, 4},
5926 {0x01b6, 0x01b9, 3},
5927 {0x01bd, 0x01bf, 2},
5928 {0x01c5, 0x01c6, 1},
5929 {0x01c8, 0x01c9, 1},
5930 {0x01cb, 0x01cc, 1},
5931 {0x01ce, 0x01dc, 2},
5932 {0x01dd, 0x01ef, 2},
5933 {0x01f2, 0x01f3, 1},
5934 {0x01f5, 0x01f9, 4},
5935 {0x01fb, 0x021f, 2},
5936 {0x0223, 0x0233, 2},
5937 {0x023c, 0x023f, 3},
5938 {0x0240, 0x0242, 2},
5939 {0x0247, 0x024f, 2},
5940 {0x0250, 0x0254, 1},
5941 {0x0256, 0x0257, 1},
5942 {0x0259, 0x025b, 2},
5943 {0x0260, 0x0263, 3},
5944 {0x0265, 0x0268, 3},
5945 {0x0269, 0x026b, 2},
5946 {0x026f, 0x0271, 2},
5947 {0x0272, 0x0275, 3},
5948 {0x027d, 0x0283, 3},
5949 {0x0288, 0x028c, 1},
5950 {0x0292, 0x0345, 179},
5951 {0x0371, 0x0373, 2},
5952 {0x0377, 0x037b, 4},
5953 {0x037c, 0x037d, 1},
5954 {0x03ac, 0x03af, 1},
5955 {0x03b1, 0x03ce, 1},
5956 {0x03d0, 0x03d1, 1},
5957 {0x03d5, 0x03d7, 1},
5958 {0x03d9, 0x03ef, 2},
5959 {0x03f0, 0x03f2, 1},
5960 {0x03f5, 0x03fb, 3},
5961 {0x0430, 0x045f, 1},
5962 {0x0461, 0x0481, 2},
5963 {0x048b, 0x04bf, 2},
5964 {0x04c2, 0x04ce, 2},

```

5965         {0x04cf, 0x0527, 2},
5966         {0x0561, 0x0586, 1},
5967         {0x1d79, 0x1d7d, 4},
5968         {0x1e01, 0x1e95, 2},
5969         {0x1e9b, 0x1ea1, 6},
5970         {0x1ea3, 0x1eff, 2},
5971         {0x1f00, 0x1f07, 1},
5972         {0x1f10, 0x1f15, 1},
5973         {0x1f20, 0x1f27, 1},
5974         {0x1f30, 0x1f37, 1},
5975         {0x1f40, 0x1f45, 1},
5976         {0x1f51, 0x1f57, 2},
5977         {0x1f60, 0x1f67, 1},
5978         {0x1f70, 0x1f7d, 1},
5979         {0x1fb0, 0x1fb1, 1},
5980         {0x1fbe, 0x1fd0, 18},
5981         {0x1fd1, 0x1fe0, 15},
5982         {0x1fe1, 0x1fe5, 4},
5983         {0x214e, 0x2184, 54},
5984         {0x2c30, 0x2c5e, 1},
5985         {0x2c61, 0x2c65, 4},
5986         {0x2c66, 0x2c6c, 2},
5987         {0x2c73, 0x2c76, 3},
5988         {0x2c81, 0x2ce3, 2},
5989         {0x2cec, 0x2cee, 2},
5990         {0x2d00, 0x2d25, 1},
5991         {0xa641, 0xa66d, 2},
5992         {0xa681, 0xa697, 2},
5993         {0xa723, 0xa72f, 2},
5994         {0xa733, 0xa76f, 2},
5995         {0xa77a, 0xa77c, 2},
5996         {0xa77f, 0xa787, 2},
5997         {0xa78c, 0xa791, 5},
5998         {0xa7a1, 0xa7a9, 2},
5999         {0xff41, 0xff5a, 1},
6000     },
6001     R32: []Range32{
6002         {0x10428, 0x1044f, 1},
6003     },
6004 }
6005
6006 var foldM = &RangeTable{
6007     R16: []Range16{
6008         {0x0399, 0x03b9, 32},
6009         {0x1fbe, 0x1fbe, 1},
6010     },
6011 }
6012
6013 var foldMn = &RangeTable{
6014     R16: []Range16{

```

```
6015             {0x0399, 0x03b9, 32},
6016             {0x1fbe, 0x1fbe, 1},
6017         },
6018     },
6019
6020 // FoldScript maps a script name to a table of
6021 // code points outside the script that are equivalent under
6022 // simple case folding to code points inside the script.
6023 // If there is no entry for a script name, there are no such
6024 var FoldScript = map[string]*RangeTable{}
6025
6026 // Range entries: 3391 16-bit, 659 32-bit, 4050 total.
6027 // Range bytes: 20346 16-bit, 7908 32-bit, 28254 total.
6028
6029 // Fold orbit bytes: 63 pairs, 252 bytes
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/unicode/utf16/utf16.go

```
1 // Copyright 2010 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package utf16 implements encoding and decoding of UTF-16
6 package utf16
7
8 // The conditions replacementChar==unicode.ReplacementChar a
9 // maxRune==unicode.MaxRune are verified in the tests.
10 // Defining them locally avoids this package depending on pa
11
12 const (
13     replacementChar = '\uFFFD' // Unicode replacemen
14     maxRune         = '\U0010FFFF' // Maximum valid Unic
15 )
16
17 const (
18     // 0xd800-0xdc00 encodes the high 10 bits of a pair.
19     // 0xdc00-0xe000 encodes the low 10 bits of a pair.
20     // the value is those 20 bits plus 0x10000.
21     surr1 = 0xd800
22     surr2 = 0xdc00
23     surr3 = 0xe000
24
25     surrSelf = 0x10000
26 )
27
28 // IsSurrogate returns true if the specified Unicode code po
29 // can appear in a surrogate pair.
30 func IsSurrogate(r rune) bool {
31     return surr1 <= r && r < surr3
32 }
33
34 // DecodeRune returns the UTF-16 decoding of a surrogate pai
35 // If the pair is not a valid UTF-16 surrogate pair, Decoder
36 // the Unicode replacement code point U+FFFD.
37 func DecodeRune(r1, r2 rune) rune {
38     if surr1 <= r1 && r1 < surr2 && surr2 <= r2 && r2 <
39         return (rune(r1)-surr1)<<10 | (rune(r2) - su
40     }
41     return replacementChar
```

```

42 }
43
44 // EncodeRune returns the UTF-16 surrogate pair r1, r2 for t
45 // If the rune is not a valid Unicode code point or does not
46 // EncodeRune returns U+FFFD, U+FFFD.
47 func EncodeRune(r rune) (r1, r2 rune) {
48     if r < surrSelf || r > maxRune || IsSurrogate(r) {
49         return replacementChar, replacementChar
50     }
51     r -= surrSelf
52     return surr1 + (r>>10)&0x3ff, surr2 + r&0x3ff
53 }
54
55 // Encode returns the UTF-16 encoding of the Unicode code po
56 func Encode(s []rune) []uint16 {
57     n := len(s)
58     for _, v := range s {
59         if v >= surrSelf {
60             n++
61         }
62     }
63
64     a := make([]uint16, n)
65     n = 0
66     for _, v := range s {
67         switch {
68             case v < 0, surr1 <= v && v < surr3, v > max
69                 v = replacementChar
70                 fallthrough
71             case v < surrSelf:
72                 a[n] = uint16(v)
73                 n++
74             default:
75                 r1, r2 := EncodeRune(v)
76                 a[n] = uint16(r1)
77                 a[n+1] = uint16(r2)
78                 n += 2
79         }
80     }
81     return a[0:n]
82 }
83
84 // Decode returns the Unicode code point sequence represente
85 // by the UTF-16 encoding s.
86 func Decode(s []uint16) []rune {
87     a := make([]rune, len(s))
88     n := 0
89     for i := 0; i < len(s); i++ {
90         switch r := s[i]; {
91             case surr1 <= r && r < surr2 && i+1 < len(s)

```

```
92             surr2 <= s[i+1] && s[i+1] < surr3:
93             // valid surrogate sequence
94             a[n] = DecodeRune(rune(r), rune(s[i+
95             i++
96             n++
97         case surr1 <= r && r < surr3:
98             // invalid surrogate sequence
99             a[n] = replacementChar
100            n++
101        default:
102            // normal rune
103            a[n] = rune(r)
104            n++
105        }
106    }
107    return a[0:n]
108 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file

src/pkg/unicode/utf8/utf8.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package utf8 implements functions and constants to support
6 // UTF-8. It includes functions to translate between runes a
7 package utf8
8
9 // The conditions RuneError==unicode.ReplacementChar and
10 // MaxRune==unicode.MaxRune are verified in the tests.
11 // Defining them locally avoids this package depending on pa
12
13 // Numbers fundamental to the encoding.
14 const (
15     RuneError = '\uFFFD' // the "error" Rune or "Uni
16     RuneSelf  = 0x80     // characters below Runesel
17     MaxRune   = '\U0010FFFF' // Maximum valid Unicode co
18     UTFMax    = 4        // maximum number of bytes
19 )
20
21 const (
22     t1 = 0x00 // 0000 0000
23     tx = 0x80 // 1000 0000
24     t2 = 0xC0 // 1100 0000
25     t3 = 0xE0 // 1110 0000
26     t4 = 0xF0 // 1111 0000
27     t5 = 0xF8 // 1111 1000
28
29     maskx = 0x3F // 0011 1111
30     mask2 = 0x1F // 0001 1111
31     mask3 = 0x0F // 0000 1111
32     mask4 = 0x07 // 0000 0111
33
34     rune1Max = 1<<7 - 1
35     rune2Max = 1<<11 - 1
36     rune3Max = 1<<16 - 1
37     rune4Max = 1<<21 - 1
38 )
39
40 func decodeRuneInternal(p []byte) (r rune, size int, short b
41     n := len(p)
```

```

42     if n < 1 {
43         return RuneError, 0, true
44     }
45     c0 := p[0]
46
47     // 1-byte, 7-bit sequence?
48     if c0 < tx {
49         return rune(c0), 1, false
50     }
51
52     // unexpected continuation byte?
53     if c0 < t2 {
54         return RuneError, 1, false
55     }
56
57     // need first continuation byte
58     if n < 2 {
59         return RuneError, 1, true
60     }
61     c1 := p[1]
62     if c1 < tx || t2 <= c1 {
63         return RuneError, 1, false
64     }
65
66     // 2-byte, 11-bit sequence?
67     if c0 < t3 {
68         r = rune(c0&mask2)<<6 | rune(c1&maskx)
69         if r <= rune1Max {
70             return RuneError, 1, false
71         }
72         return r, 2, false
73     }
74
75     // need second continuation byte
76     if n < 3 {
77         return RuneError, 1, true
78     }
79     c2 := p[2]
80     if c2 < tx || t2 <= c2 {
81         return RuneError, 1, false
82     }
83
84     // 3-byte, 16-bit sequence?
85     if c0 < t4 {
86         r = rune(c0&mask3)<<12 | rune(c1&maskx)<<6 |
87         if r <= rune2Max {
88             return RuneError, 1, false
89         }
90         return r, 3, false
91     }

```

```

92
93     // need third continuation byte
94     if n < 4 {
95         return RuneError, 1, true
96     }
97     c3 := p[3]
98     if c3 < tx || t2 <= c3 {
99         return RuneError, 1, false
100    }
101
102    // 4-byte, 21-bit sequence?
103    if c0 < t5 {
104        r = rune(c0&mask4)<<18 | rune(c1&maskx)<<12
105        if r <= rune3Max {
106            return RuneError, 1, false
107        }
108        return r, 4, false
109    }
110
111    // error
112    return RuneError, 1, false
113 }
114
115 func decodeRuneInStringInternal(s string) (r rune, size int,
116     n := len(s)
117     if n < 1 {
118         return RuneError, 0, true
119     }
120     c0 := s[0]
121
122     // 1-byte, 7-bit sequence?
123     if c0 < tx {
124         return rune(c0), 1, false
125     }
126
127     // unexpected continuation byte?
128     if c0 < t2 {
129         return RuneError, 1, false
130     }
131
132     // need first continuation byte
133     if n < 2 {
134         return RuneError, 1, true
135     }
136     c1 := s[1]
137     if c1 < tx || t2 <= c1 {
138         return RuneError, 1, false
139     }
140

```

```

141 // 2-byte, 11-bit sequence?
142 if c0 < t3 {
143     r = rune(c0&mask2)<<6 | rune(c1&maskx)
144     if r <= rune1Max {
145         return RuneError, 1, false
146     }
147     return r, 2, false
148 }
149
150 // need second continuation byte
151 if n < 3 {
152     return RuneError, 1, true
153 }
154 c2 := s[2]
155 if c2 < tx || t2 <= c2 {
156     return RuneError, 1, false
157 }
158
159 // 3-byte, 16-bit sequence?
160 if c0 < t4 {
161     r = rune(c0&mask3)<<12 | rune(c1&maskx)<<6 |
162     if r <= rune2Max {
163         return RuneError, 1, false
164     }
165     return r, 3, false
166 }
167
168 // need third continuation byte
169 if n < 4 {
170     return RuneError, 1, true
171 }
172 c3 := s[3]
173 if c3 < tx || t2 <= c3 {
174     return RuneError, 1, false
175 }
176
177 // 4-byte, 21-bit sequence?
178 if c0 < t5 {
179     r = rune(c0&mask4)<<18 | rune(c1&maskx)<<12
180     if r <= rune3Max {
181         return RuneError, 1, false
182     }
183     return r, 4, false
184 }
185
186 // error
187 return RuneError, 1, false
188 }
189

```

```

190 // FullRune reports whether the bytes in p begin with a full
191 // An invalid encoding is considered a full Rune since it wi
192 func FullRune(p []byte) bool {
193     _, _, short := decodeRuneInternal(p)
194     return !short
195 }
196
197 // FullRuneInString is like FullRune but its input is a stri
198 func FullRuneInString(s string) bool {
199     _, _, short := decodeRuneInStringInternal(s)
200     return !short
201 }
202
203 // DecodeRune unpacks the first UTF-8 encoding in p and retu
204 // If the encoding is invalid, it returns (RuneError, 1), an
205 func DecodeRune(p []byte) (r rune, size int) {
206     r, size, _ = decodeRuneInternal(p)
207     return
208 }
209
210 // DecodeRuneInString is like DecodeRune but its input is a
211 // If the encoding is invalid, it returns (RuneError, 1), an
212 func DecodeRuneInString(s string) (r rune, size int) {
213     r, size, _ = decodeRuneInStringInternal(s)
214     return
215 }
216
217 // DecodeLastRune unpacks the last UTF-8 encoding in p and r
218 // If the encoding is invalid, it returns (RuneError, 1), an
219 func DecodeLastRune(p []byte) (r rune, size int) {
220     end := len(p)
221     if end == 0 {
222         return RuneError, 0
223     }
224     start := end - 1
225     r = rune(p[start])
226     if r < RuneSelf {
227         return r, 1
228     }
229     // guard against O(n^2) behavior when traversing
230     // backwards through strings with long sequences of
231     // invalid UTF-8.
232     lim := end - UTFMax
233     if lim < 0 {
234         lim = 0
235     }
236     for start--; start >= lim; start-- {
237         if RuneStart(p[start]) {
238             break
239         }

```

```

240     }
241     if start < 0 {
242         start = 0
243     }
244     r, size = DecodeRune(p[start:end])
245     if start+size != end {
246         return RuneError, 1
247     }
248     return r, size
249 }
250
251 // DecodeLastRuneInString is like DecodeLastRune but its inp
252 // If the encoding is invalid, it returns (RuneError, 1), an
253 func DecodeLastRuneInString(s string) (r rune, size int) {
254     end := len(s)
255     if end == 0 {
256         return RuneError, 0
257     }
258     start := end - 1
259     r = rune(s[start])
260     if r < RuneSelf {
261         return r, 1
262     }
263     // guard against O(n^2) behavior when traversing
264     // backwards through strings with long sequences of
265     // invalid UTF-8.
266     lim := end - UTFMax
267     if lim < 0 {
268         lim = 0
269     }
270     for start--; start >= lim; start-- {
271         if RuneStart(s[start]) {
272             break
273         }
274     }
275     if start < 0 {
276         start = 0
277     }
278     r, size = DecodeRuneInString(s[start:end])
279     if start+size != end {
280         return RuneError, 1
281     }
282     return r, size
283 }
284
285 // RuneLen returns the number of bytes required to encode th
286 func RuneLen(r rune) int {
287     switch {
288     case r <= rune1Max:

```

```

289         return 1
290     case r <= rune2Max:
291         return 2
292     case r <= rune3Max:
293         return 3
294     case r <= rune4Max:
295         return 4
296     }
297     return -1
298 }
299
300 // EncodeRune writes into p (which must be large enough) the
301 // It returns the number of bytes written.
302 func EncodeRune(p []byte, r rune) int {
303     // Negative values are erroneous. Making it unsigne
304     if uint32(r) <= rune1Max {
305         p[0] = byte(r)
306         return 1
307     }
308
309     if uint32(r) <= rune2Max {
310         p[0] = t2 | byte(r>>6)
311         p[1] = tx | byte(r)&maskx
312         return 2
313     }
314
315     if uint32(r) > MaxRune {
316         r = RuneError
317     }
318
319     if uint32(r) <= rune3Max {
320         p[0] = t3 | byte(r>>12)
321         p[1] = tx | byte(r>>6)&maskx
322         p[2] = tx | byte(r)&maskx
323         return 3
324     }
325
326     p[0] = t4 | byte(r>>18)
327     p[1] = tx | byte(r>>12)&maskx
328     p[2] = tx | byte(r>>6)&maskx
329     p[3] = tx | byte(r)&maskx
330     return 4
331 }
332
333 // RuneCount returns the number of runes in p. Erroneous an
334 // encodings are treated as single runes of width 1 byte.
335 func RuneCount(p []byte) int {
336     i := 0
337     var n int

```



```
388         // value encoded properly. Decode it
389         // it's the 1 byte sentinel value.
390         _, size := DecodeRuneInString(s[i:])
391         if size == 1 {
392             return false
393         }
394     }
395 }
396 return true
397 }
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)

Source file src/pkg/unsafe/unsafe.go

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 /*
6      Package unsafe contains operations that step around
7 */
8 package unsafe
9
10 // ArbitraryType is here for the purposes of documentation o
11 // part of the unsafe package. It represents the type of an
12 type ArbitraryType int
13
14 // Pointer represents a pointer to an arbitrary type. There
15 // available for type Pointer that are not available for oth
16 //      1) A pointer value of any type can be converted to a
17 //      2) A Pointer can be converted to a pointer value of
18 //      3) A uintptr can be converted to a Pointer.
19 //      4) A Pointer can be converted to a uintptr.
20 // Pointer therefore allows a program to defeat the type sys
21 // arbitrary memory. It should be used with extreme care.
22 type Pointer *ArbitraryType
23
24 // Sizeof returns the size in bytes occupied by the value v.
25 // "top level" of the value only. For instance, if v is a s
26 // the slice descriptor, not the size of the memory referenc
27 func Sizeof(v ArbitraryType) uintptr
28
29 // Offsetof returns the offset within the struct of the fiel
30 // which must be of the form structValue.field. In other wo
31 // number of bytes between the start of the struct and the s
32 func Offsetof(v ArbitraryType) uintptr
33
34 // Alignof returns the alignment of the value v. It is the
35 // that the address of a variable with the type of v will al
36 // If v is of the form structValue.field, it returns the ali
37 func Alignof(v ArbitraryType) uintptr
```

Build version go1.0.1.

Except as [noted](#), the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code is licensed under a [BSD license](#).
[Terms of Service](#) | [Privacy Policy](#)